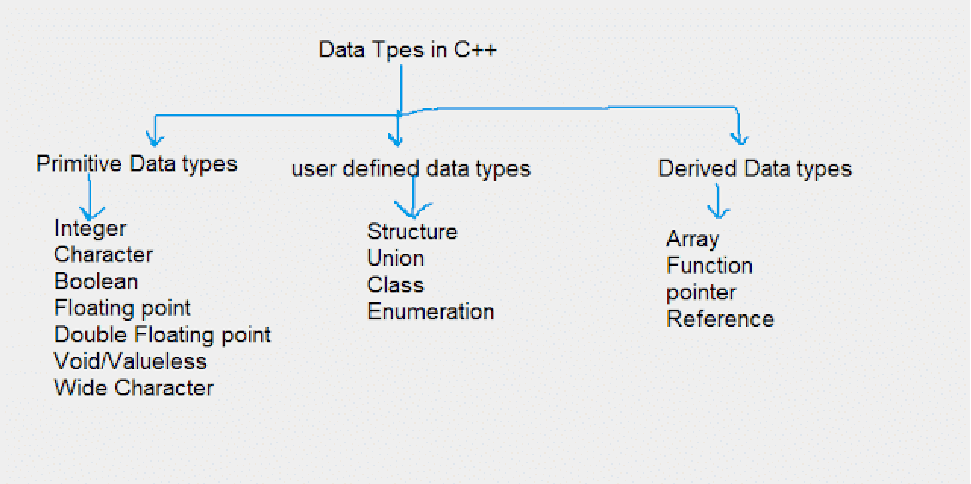
Ch.5

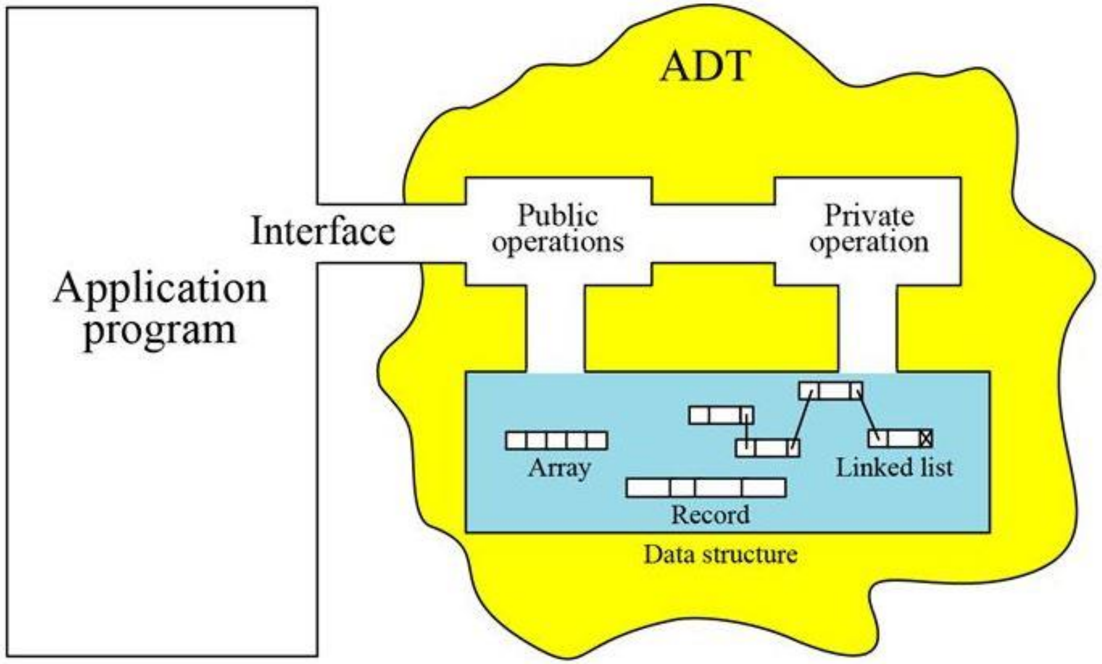
Data Structure and

Abstract Data Type

Computer Science, KMITL

@cskmitl · College & University

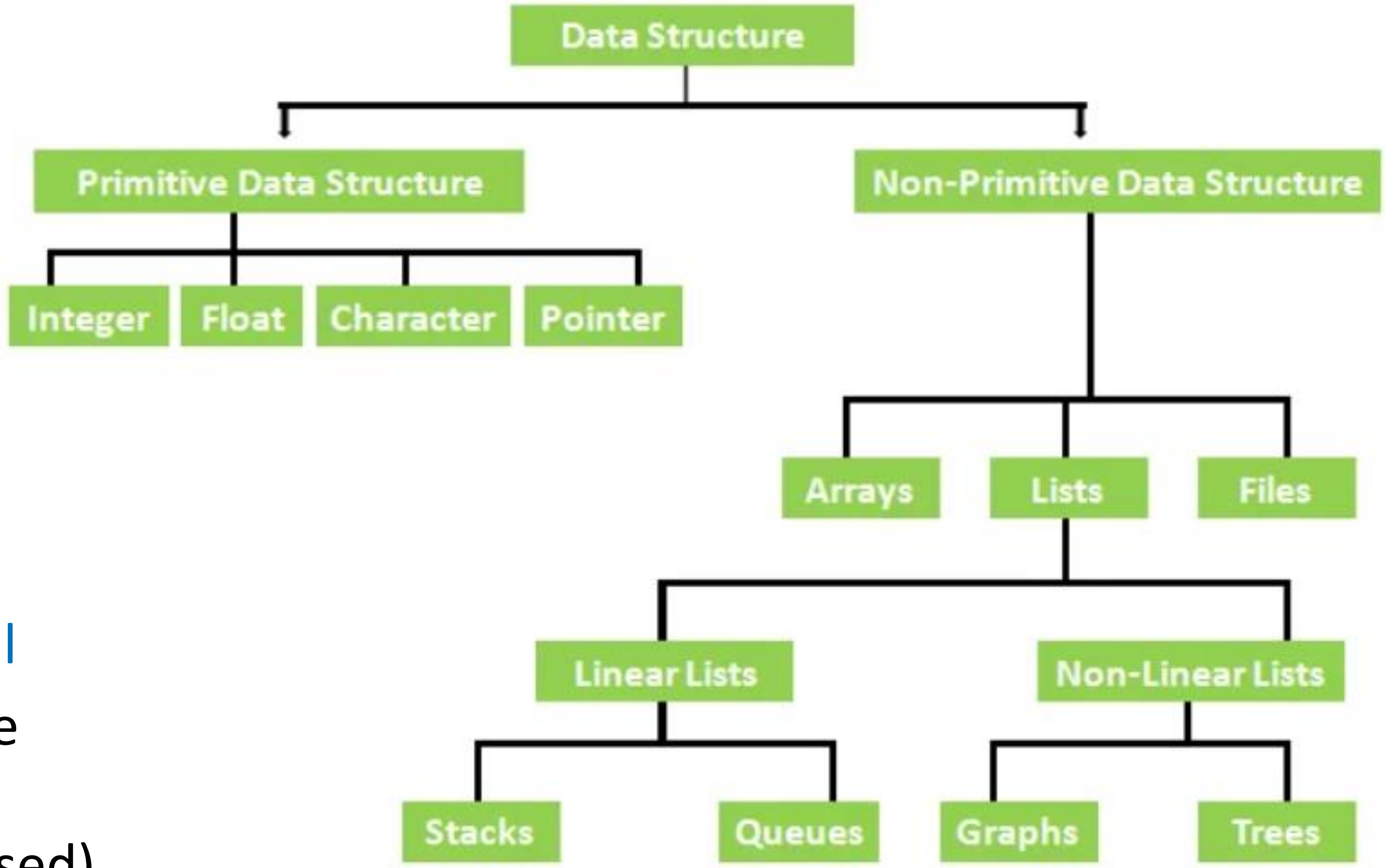




https://sarick.me/2016/05/17/a-brief-intro-to-abstract-data-types/

Outline

- (Array vs.) ArrayList
- Node
  - Linked List
- (ADT) Queue and Stack
- BTreeNode
  - Binary Search Tree
  - Tree InOrder Traversal
- Example Stack and Queue Application (BFS, DFS)
- Circular Queue (Array-based)
- Misc



https://www.tutorialscan.com/data_structure/classification-of-data-structure/

```java
static void demo1() {
  ArrayList<Integer> list = new ArrayList<>();
  list.add(1);
  list.add(2);
  System.out.print("[");
  for (int n : list)
    System.out.print(n + " ");
  System.out.println("]");    // [1 2]

  int ans =  list.remove(0);
  System.out.print("ans = " + ans + " -> [");
  for (int n : list)
    System.out.print(n + " ");
  System.out.println("]");    // [2]

  list.add(5);  list.add(3);
  list.add(3,1); //2 5 '1' 3 4
  list.add(6);  list.add(4);
  System.out.print("[");
  for (int i = 0; i < list.size(); i++)
    System.out.print(list.get(i) + " ");
  System.out.print("] (");
  System.out.println(list.size()+")");
  // [2 5 3 1 6 4 ] (6)
}
```

Methods	Description
boolean add (E obj)	Adds element at last. Returns true if added else false.
Void add (int pos, E obj)	Inserts element at a specific position.
E remove (int pos)	Removes element and returns its reference.
Void clear()	Removes all elements from list
E set (int pos, E obj)	Replaces the existing element with new element.
boolean contains (E obj)	Returns **true** if element exists, else **false**.
E get (int pos)	Returns element at specified position.
Int indexOf(E obj)	Returns position of specified object.
Int lastindexOf(E obj)	Returns position of last occurrence of specified element.
Int size()	Returns number of elements it contains.
Object[] to Array()	Returns an Object class type array containing all elements.

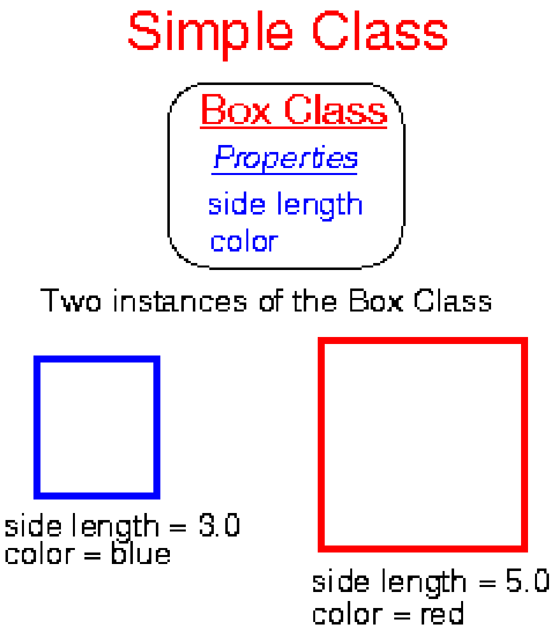
https://realjavaonline.com/coll-frame/pics/table.PNG

```java
static void demo1_array() {
  int [] arr = new int[6];
    arr[0] = 2;  arr[1] = 5;
    arr[2] = 3;  arr[3] = 1;
    arr[4] = 6;  arr[5] = 4;
  System.out.print("[");
  for (int i = 0; i < arr.length; i++)
      System.out.print(arr[i] + " ");
  System.out.print("] (");
  System.out.println(arr.length+")");
  // [2 5 3 1 6 4 ] (6)
}
```

ArrayList of objects

```
static void demo2() {
  ArrayList<Box> list = new ArrayList<>();

  Box a_box = new Box();
  a_box.set_attributes(3, "blue");
  list.add(a_box);

  a_box = new Box();
  a_box.set_attributes(5, "red");
  list.add(a_box);

  for (Box b : list)
    println("My Type is Box(" + b.side_length
                         +", " +  b.color +")");
  // My Type is Box(3, blue)
  // My Type is Box(5, red)
}
```

```
class Box {
    int side_length;
    String color;
    void set_attributes(int len,
                     String col) {
        side_length = len;
        color = col;
    }
}
```

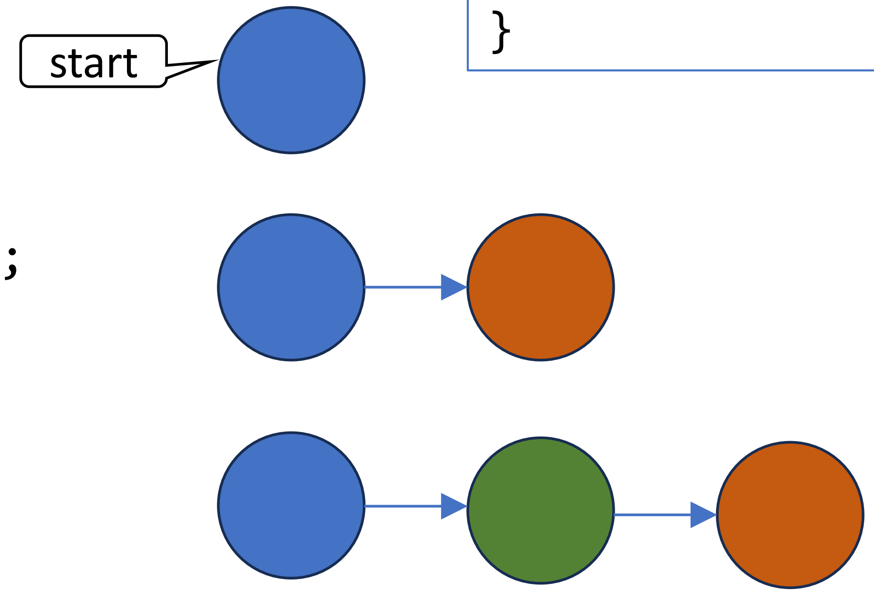


https://www.ncl.ucar.edu/Document/HLUs/User_Guide/classes/classoview.shtml

Node (and chained nodes i.e. linked list)

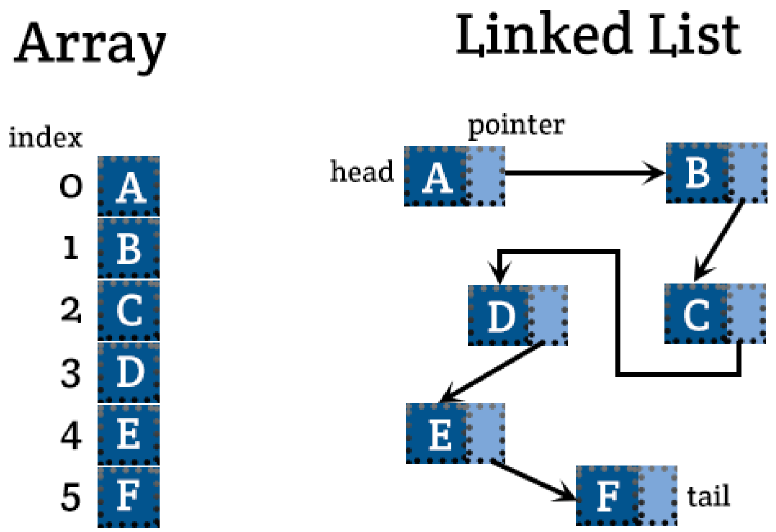
```
static void demo3() {
  Province bangkok = new Province("Bangkok");
  Province start = bangkok;
  bangkok.nextProvince = new Province("Samutsongkram");
  Province sakorn = new Province("Samutsakorn");
  sakorn.nextProvince = bangkok.nextProvince;
  bangkok.nextProvince = sakorn;

  Province city = start;
  while (city != null) {
      System.out.print(city.name + " ");
      city = city.nextProvince;
  }
  System.out.println();
  // Bangkok Samutsakorn Samutsongkram
}
```

```
class Province {
    String name;
    Province nextProvince;
    Province(String n) {
        name = n;
    }
}
```



Array vs. Linked List (Classical Data Structure)

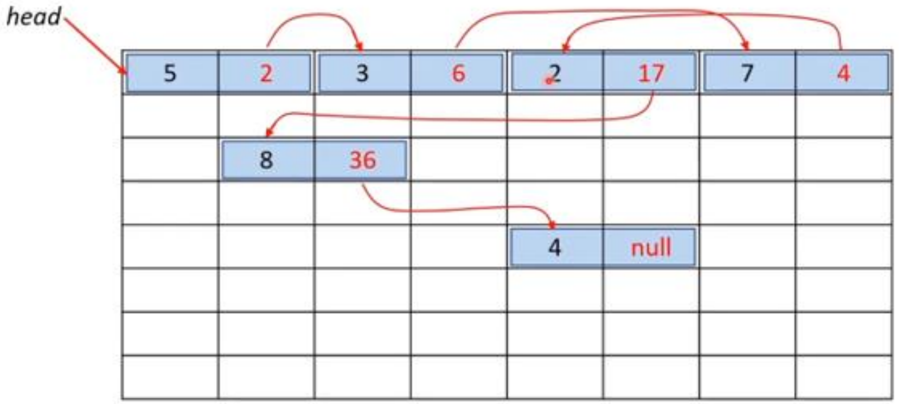
- Array stores data in a contiguous manner such that data can be accessed via an index. Its size must be presented when allocate it.
- Linked list stores data in a non-contiguous manner. Its advantage of its dynamic size is traded off with the cost of accessing a member.



	Array	Linked List
Pros	Fast Search Time (O(1)) Less memory required per element Take advantage of locality	(Detailed but) better Insertion/Deletion Time (O(n)) Fit Size Efficient memory allocation
Cons	Slow insertion/Deletion Time (O(n)) Fixed Size Inefficient memory allocation	Slow Search Time (O(n)) More memory required per element for pointer

Linked List vs. ArrayList

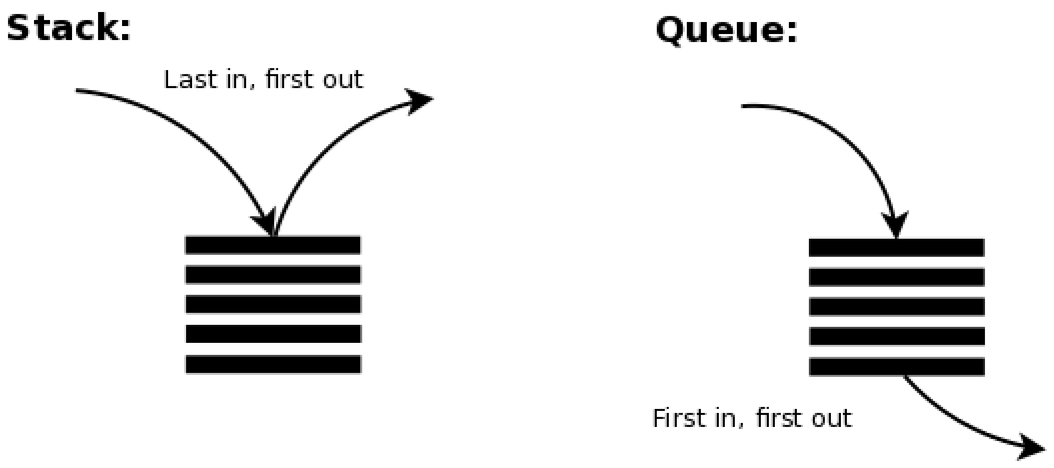
Linked List	ArrayList
Could uses **doubly Linked List** to store the elements.	Internally uses **dynamic array** to store the elements.
Manipulation with Linked List is **faster** than ArrayList because of no shifting.	Manipulation with ArrayList is **slow**. If any element is removed from the array, all the bits are shifted in memory.
Linked List is better for **manipulating** data.	ArrayList is better for **storing and accessing**. ArrayList index makes accessing element faster.
Doubly Linked List can act as a queue.	Because of the array-based underneath, manipulating like a queue is more complex.



var	address
head	0

(Early Structure vs.) Abstract Data Types

- ADT
  - A collection of data with (collection's) set of operations.
  - E.g. Stack vsQueue
    - Early days – study of implement from data structure underneath (array, linked list)
    - Today – if not present, apply existing one.



https://gohighbrow.com/stacks-and-queues/

DEQUE	STACK	QUEUE
size()	size()	size()
isEmpty()	isEmpty()	isEmpty()
Insert_First()	-	-
Insert_Last()	Push()	Enqueue()
Remove_First()	-	Dequeue()
Remove_Last()	Pop()	-

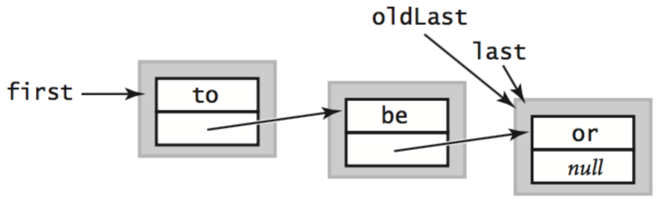
https://www.geeksforgeeks.org/implement-stack-queue-using-deque/

Queue

- A queue supports the insert and remove operations using a first-in first-out (FIFO) discipline. By convention, we name the queue insert operation enqueue and the remove operation dequeue

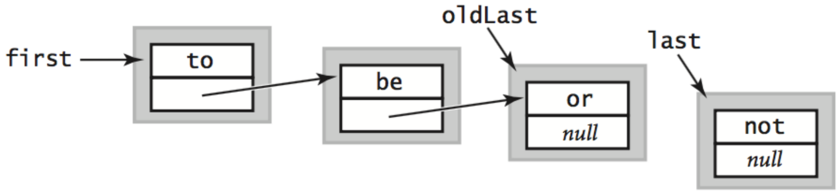
save a link to the last node

```
Node oldLast = last;
```



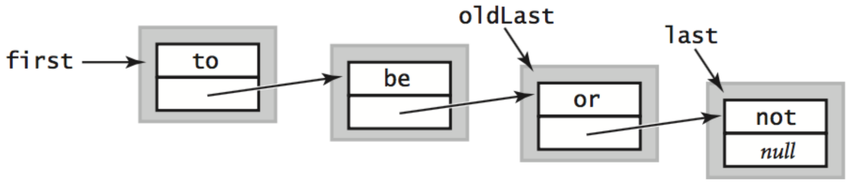
create a new node for the end

```
Node last = new Node();
last.item = "not";
```



link the new node to the end of the list

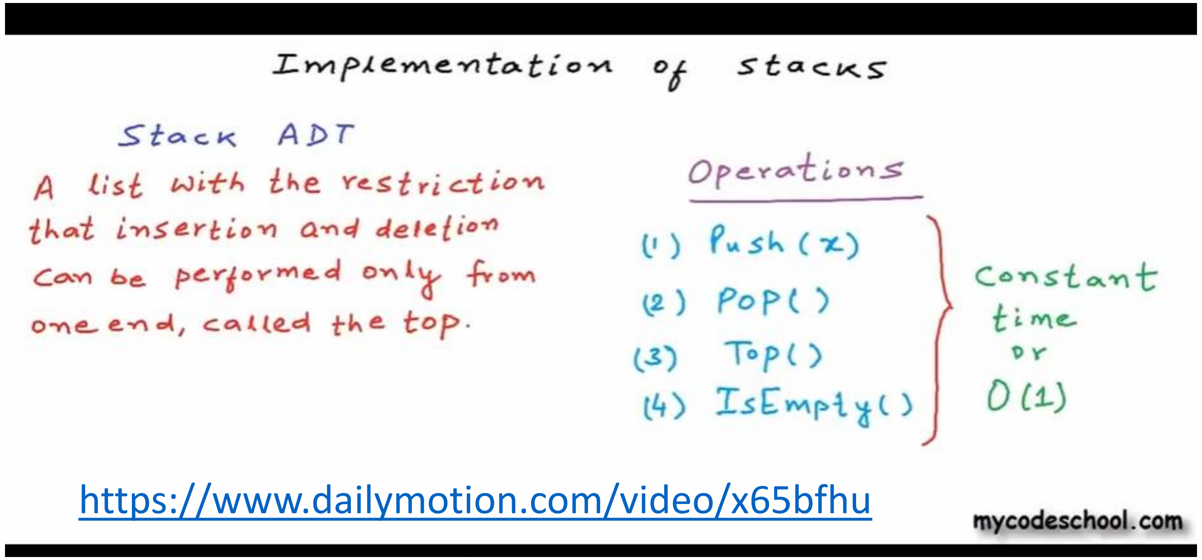
```
oldLast.next = last;
```



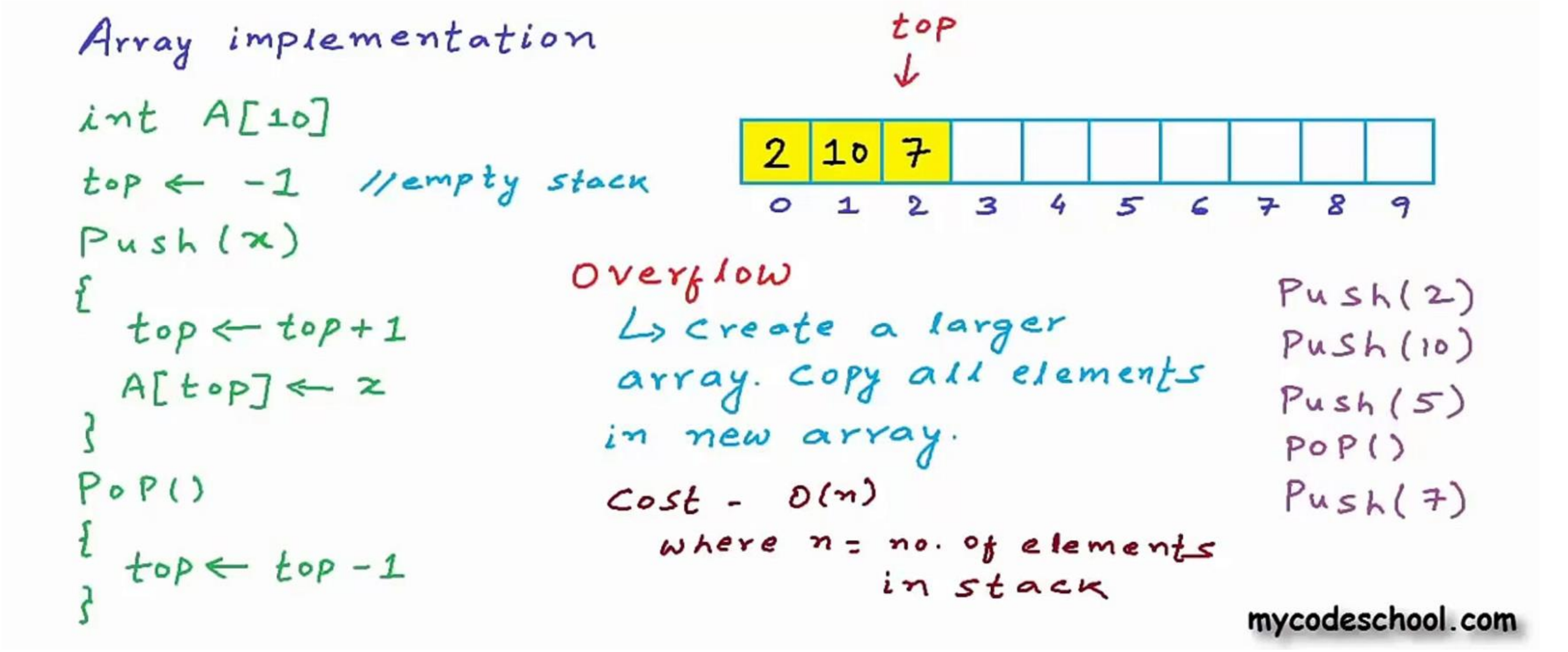
https://introcs.cs.princeton.edu/java/43stack/

Stack

- A stack is a collection that is based on the last-in-first-out (LIFO) policy. By tradition, we name the stack insert method push() and the stack remove operation pop().
    - peek()
    - isEmpty()
- Array-based (demo)



https://www.dailymotion.com/video/x65bfhu



A Statck Application

EXAMPLE: Let us illustrate the procedure *InfixToPostfix* with the following arithmetic expression:

Input: (A + B)^ C - (D * E) / F) (infix form)

Read symbol	*Stack*	*Output*
Initial	(	
1	((	
2	((	A
3	((+	A
4	((+	AB
5	(	AB+
6	(^	AB+
7	(^	AB + C
8	(–	AB + C ^
9	(– (	AB + C ^
10	(– (	AB + C ^ D
11	(– (*	AB + C ^ D
12	(– (*	AB + C ^ DE
13	(–	AB + C ^ DE *
14	(– /	AB + C ^ DE *
15	(– /	AB + C ^ DE * F
16		AB + C ^ DE * F / –

Output: A B + C ^ DE * F / – (postfix form)

https://aits-tpt.edu.in/wp-content/uploads/2018/08/DS-unit-2.1.pdf

https://www.chegg.com/homework-help/questions-and-answers/topic-data-structures-algorithms-programming-compiler-c-11-flag-static-std-c-0x-q43753642

Reverse Polish Notation

Reverse Polish Notation (RPN), also known as polish postfix notation or simply postfix notation, is a mathematical notation in which operators follow their operands.

For example, the infix expression P1: 5 + ((1 + 2) * 4) – 3 can be written like this in *Reverse Polish Notation*: P2: 5 1 2 + 4 * + 3 –

In terms of the operation, the expression P1 and P2 can be evaluated as

P1	P2
5 + ((1 + 2) * 4) – 3	5 1 2 + 4 * + 3 –
5 + (3 * 4) – 3	5 3 4 * + 3 –
5 + 12 – 3	5 12 + 3 –
17 – 3	17 3 –
14	14

The reverse polish notation has many advantages, such as there is no bracket in the expression and no priority is needed for the operators, most importantly, the evaluation process is quite simple. The reverse polish notation could be evaluated by using a stack.

Evaluation Algorithm

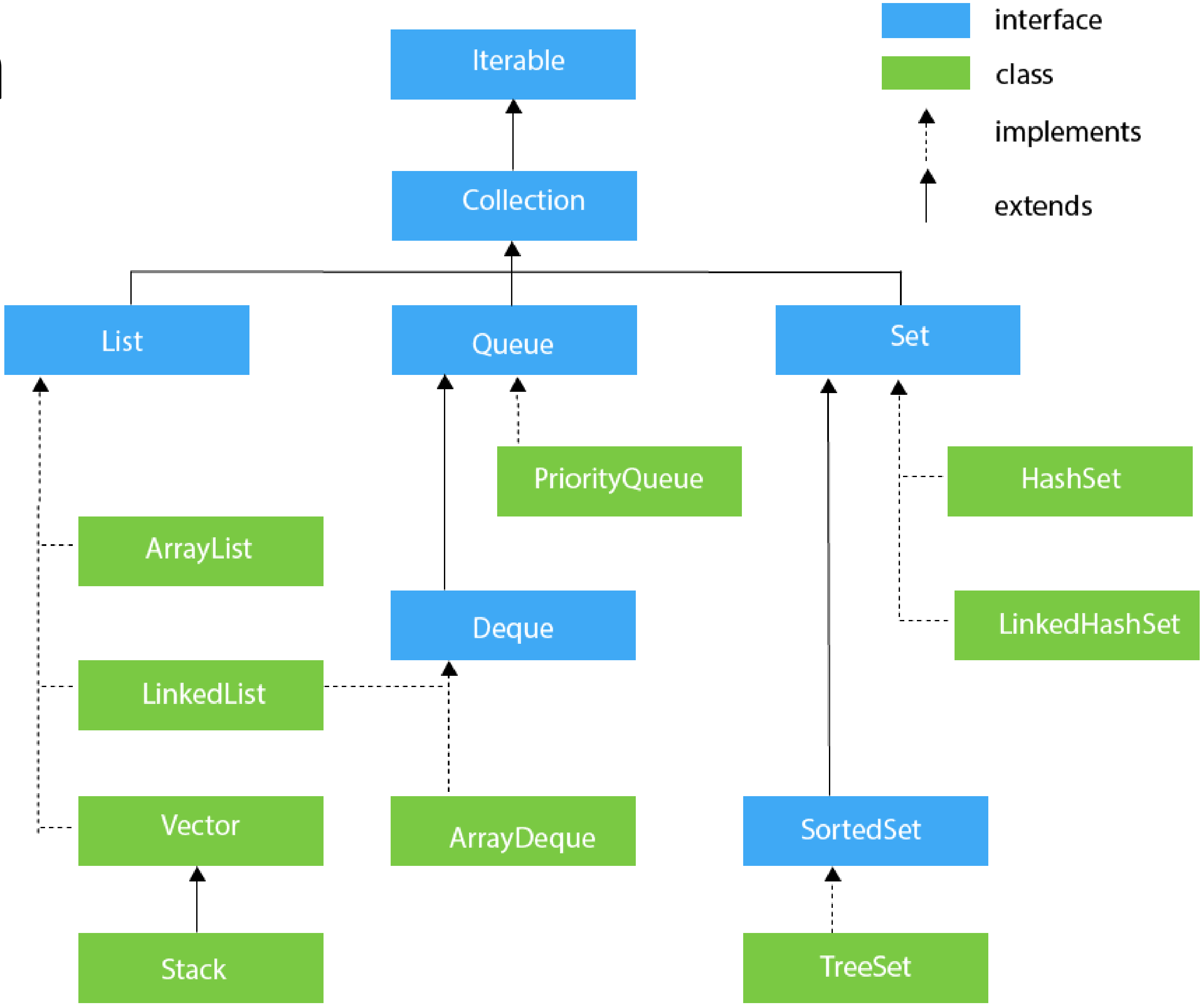
Input	Operation	Stack	Remark
5	Push	5	
1	Push	5,1	
2	Push	5,1,2	
+	**Addition**	5,3	Pop (1,2), do addition, push in the result (3)
4	Push	5,3,4	
*	**Multiplication**	5,12	Pop (3,4), do multiplication, push in the result (12)
+	**Addition**	17	Pop (5,12), do addition, push in the result (17)
3	Push	17,3	
-	**Subtraction**	14	Pop (17,3), do subtraction, push in the result (14)

collections-in-java

- Iterator interface provides the facility of iterating the elements in a forward direction only.

- Remark
  - Choice of Queue implementation (lists of methods)
  - https://www.javatpoint.com/java-arraylist
  - https://www.javatpoint.com/java-linkedlist

https://www.javatpoint.com/collections-in-java



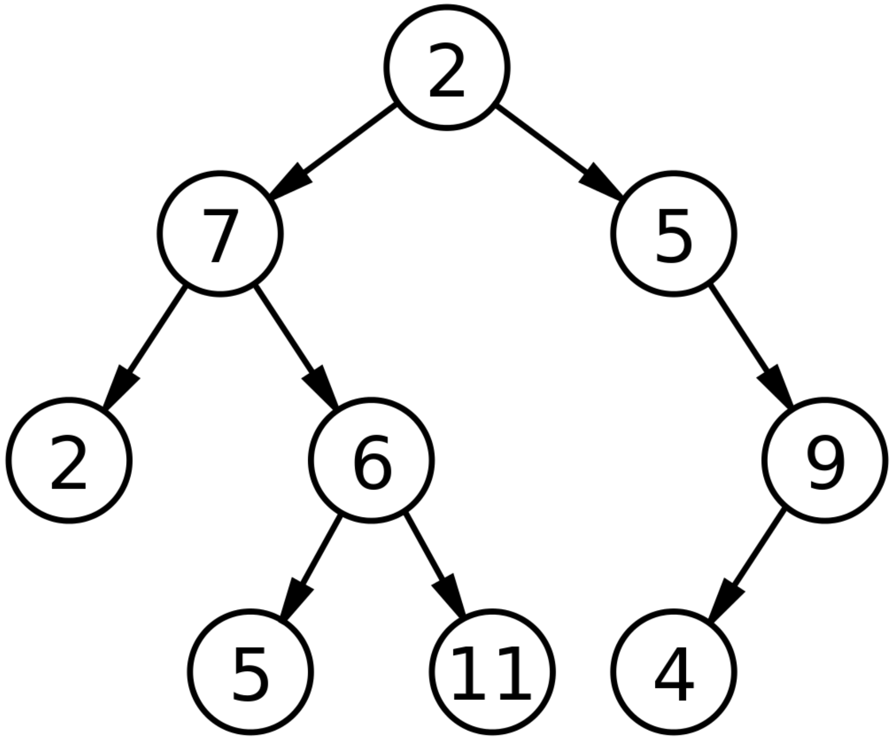
Tree

- A tree is a collection of entities called nodes. Nodes are connected by edges. Each node contains a value or data, and it may or may not have a child node .

https://www.freecodecamp.org/news/all-you-need-to-know-about-tree-data-structures-bceacb85490c/

- In computer science, a binary tree is a tree data structure in which each node has at most two children, which are referred to as the left child and the right child.

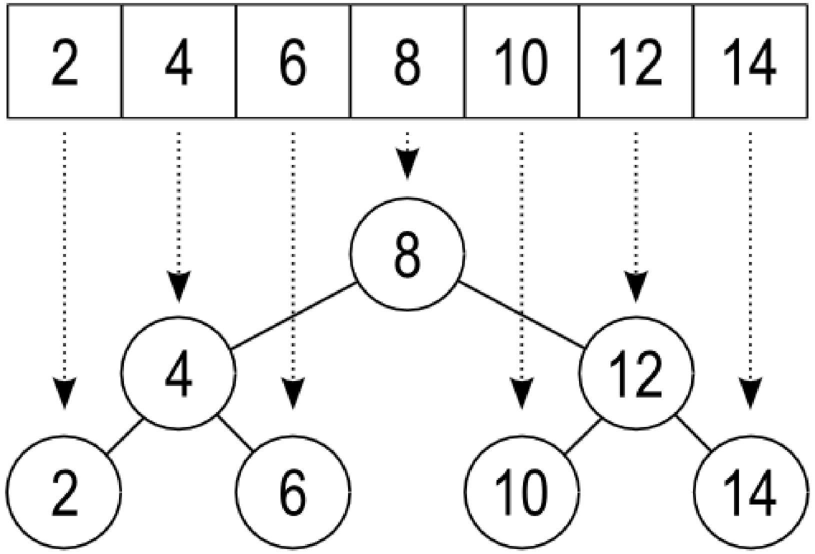
https://en.wikipedia.org/wiki/Binary_tree

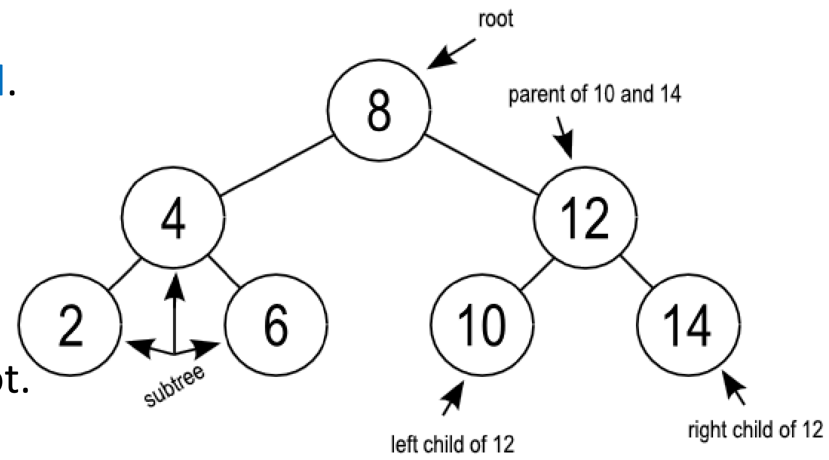


A labeled binary tree of size 9 and height 3, with a root node whose value is 2. The above tree is unbalanced.
Leaf nodes are 5, 11, and 4

BST – Binary Search Tree

- The root and all of the nodes connected to it are called a tree.
- A tree is said to be a binary tree if each element can have zero, one, or two children.
- Each element in the tree is called a node.
- The top-most node is called the root.
- Any node and all of the nodes connected below it are called a subtree.
- A node directly above another node is called its parent.
- A node directly below and to the left of another node is called its left child.
- A node directly below and to the right of another node is called its right child.
- Parent and its child is connected by an edge.
- A tree is said to be a binary search tree if every left child is smaller than its parent and every right child is larger than its parent.
- A node with no children is called a leaf.
- The height of the tree is typically defined the number of levels below the root.
- A tree is said to be perfect if for every level except the bottom level, every node has two children.



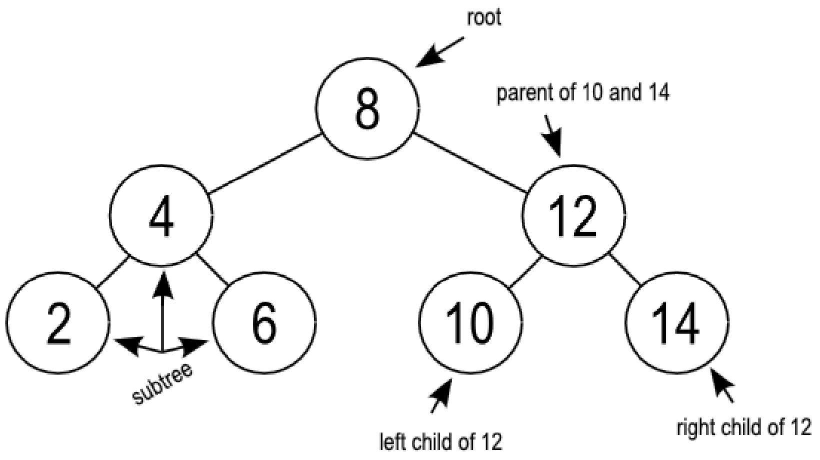


https://taylorial.com/cs2852/Bst.htm

BTreeNode

```java
static void demo4() {
  BTreeNode root = new BTreeNode(8);
  BTreeNode cur = root;
  root.left = new BTreeNode(4);
  root.right = new BTreeNode(12);
  cur = root.left;
  cur.left = new BTreeNode(2);
  cur.right = new BTreeNode(6);
  cur = root.right;
  cur.left = new BTreeNode(10);
  cur.right = new BTreeNode(14);

  demo4_inorder(root);
  System.out.println();
}
```

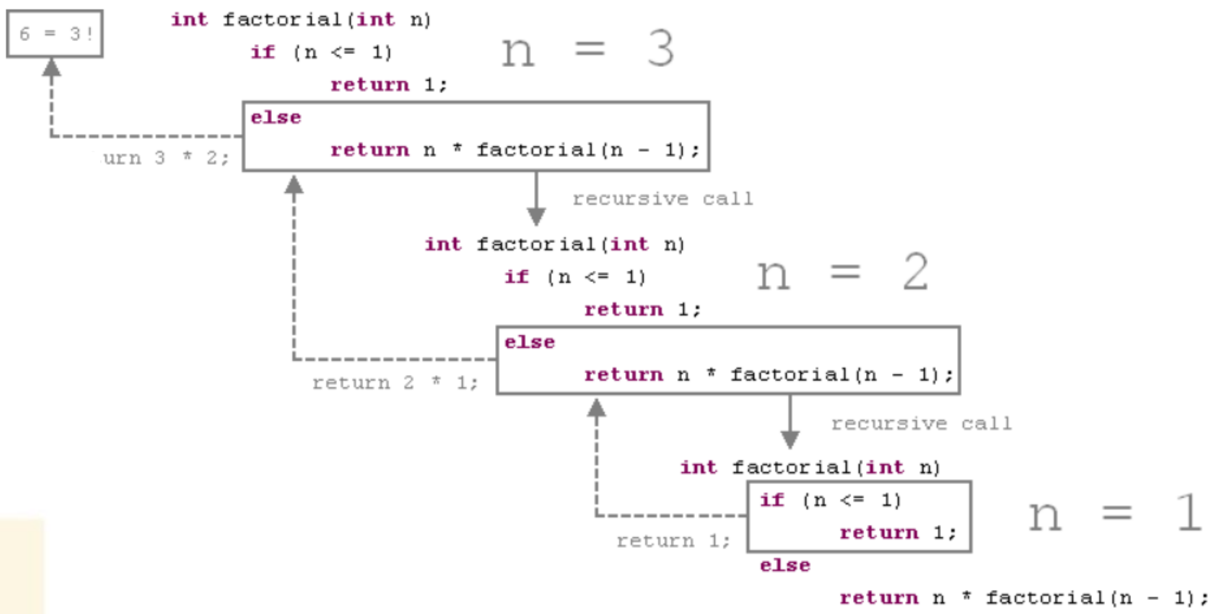
```java
class BTreeNode {
    int data;
    BTreeNode left;
    BTreeNode right;
    BTreeNode(int n) {
        data = n;
    }
}
```



Recursion (revisited)

```
int factorial(int n) {
      if (n <= 1)
            return 1;
      else
            return n * factorial(n - 1);
}
```

Calculation of 3! in details



https://www.algolist.net/Programming_concepts/Recursion

Iteration vs. Recursion

- Iteration and recursion are somewhat related
- Converting **iteration to recursion** is formulaic, but converting **recursion to iteration** can be more tricky

Iterative

```
def fact_iter(n):
    total, k = 1, 1
    while k <= n:
        total, k = total*k, k+1
    return total
```

$$n! = \prod_{k=1}^{n} k$$

Names: n, total, k, fact_iter

Recursive

```
def fact(n):
    if n == 0:
        return 1
    else:
        return n * fact(n-1)
```

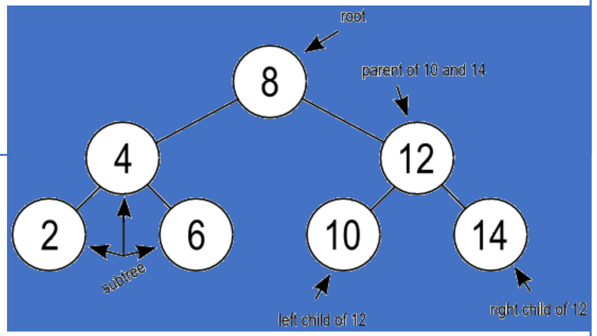
$$n! = \begin{cases} 1 & \text{if } n = 0 \\ n \cdot (n-1)! & \text{otherwise} \end{cases}$$

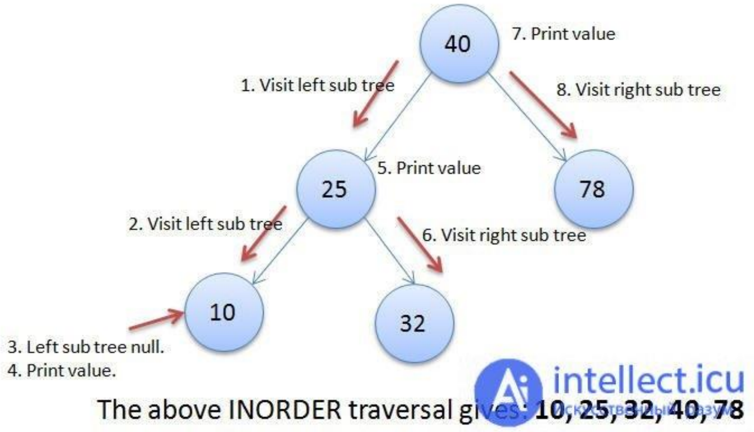
Names: n, fact

https://slideplayer.com/slide/17582544/

BST InOrder Traversing (left → Root → right)

```
            void inOrder(BNode n) {
/*10*/          if (n != null) {
/*20*/              if (n.lChild != null)
/*30*/                  inOrder(n.lChild);
/*40*/              System.out.printf("%d ", n.data);
/*50*/              if (n.rChild != null)
/*60*/                  inOrder(n.rChild);
                }
/*70*/
            }
```





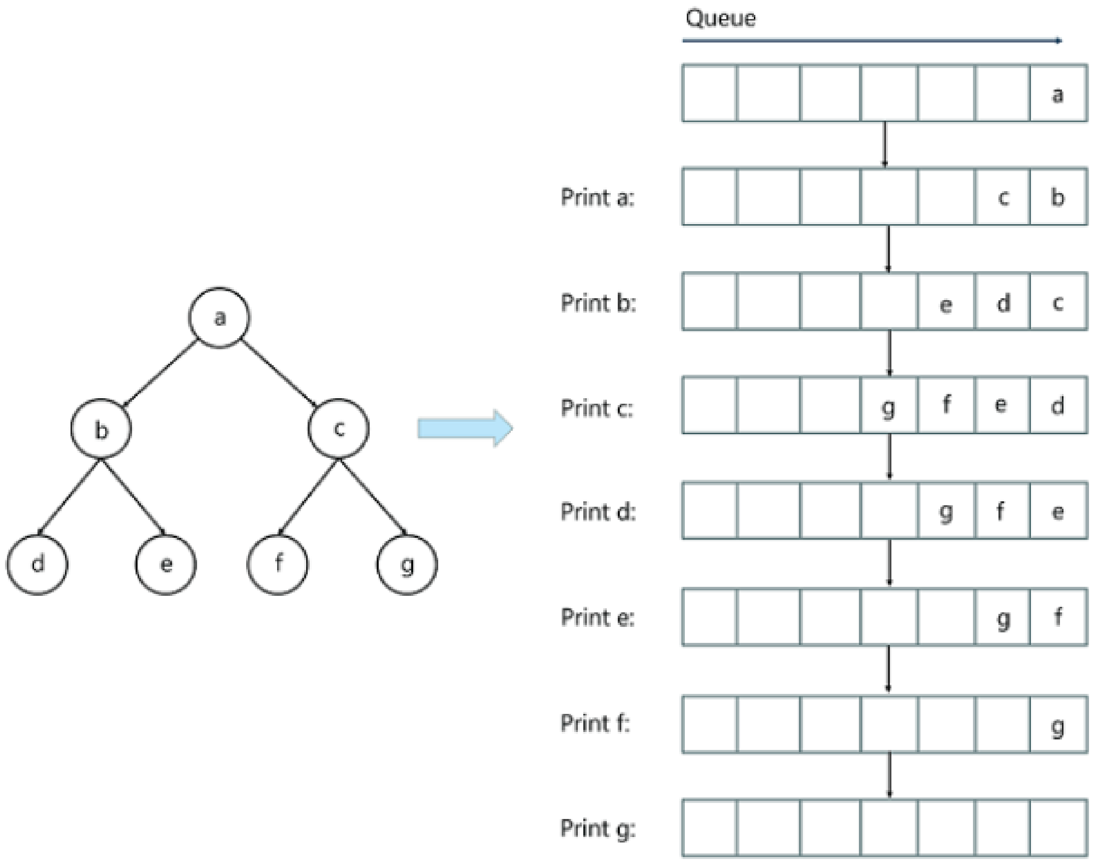
https://intellect.icu/derevo-dvoichnogo-poiska-i-obkhod-dereva-inorder-preorder-postorder-realizatsiya-4424

10 //8
30 //left
10 //4
30 //left
10 //2
20 //null
40 //print(2)
50 //null
~~70~~
40 //print(4)
60 //right
10 //6
20 //null
40 //print(6)
50 //null
~~70~~
~~70~~
40 //print(8)

60 //right
10 //12
30 //left
10 //10
20 //null
40 //print(10)
50 //null
~~70~~
40 //print(12)
60 //right
10 //14
20 //null
40 //print(14)
50 //null
~~70~~
~~70~~
~~70~~

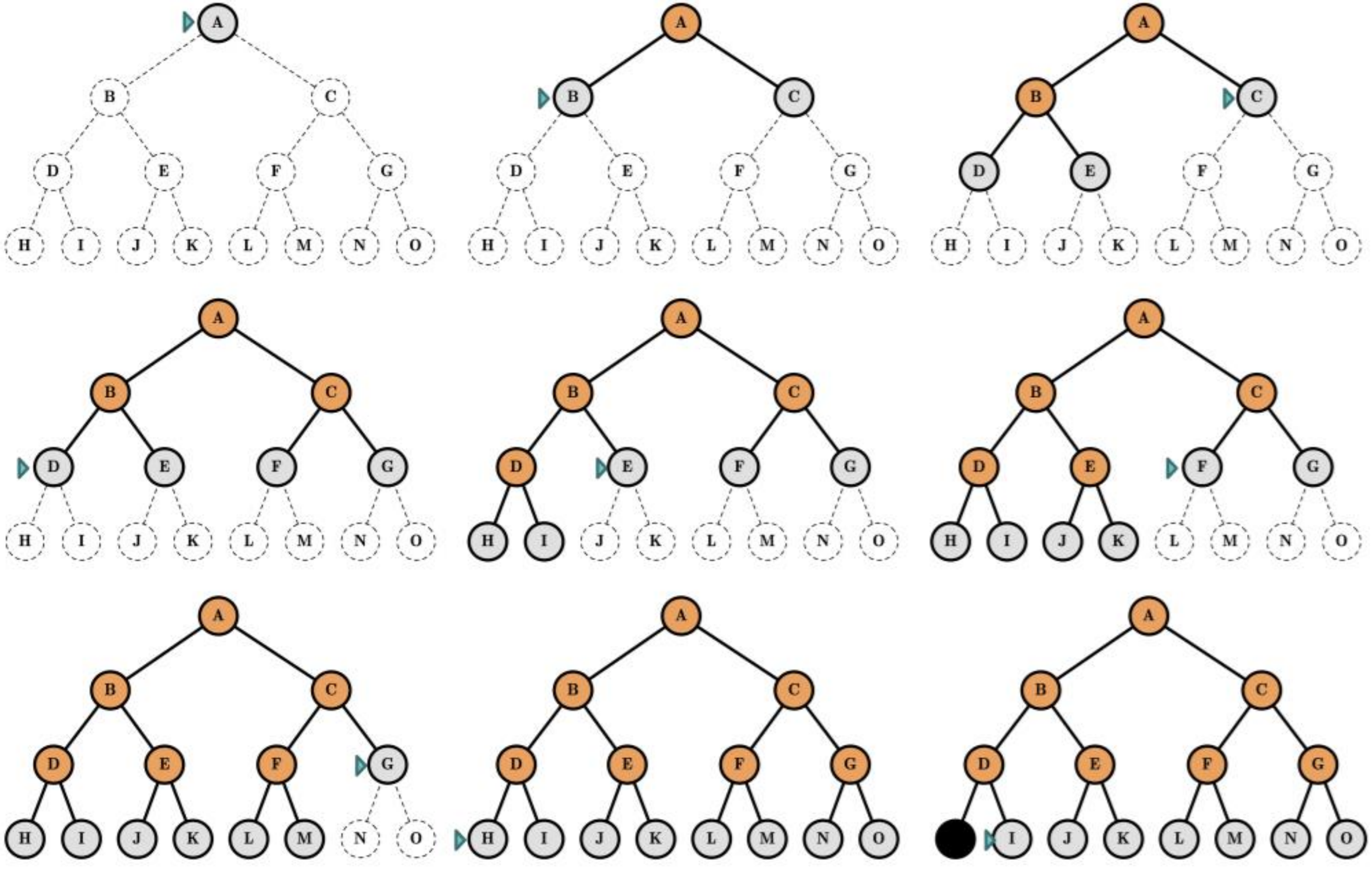
Using Queue to Perform Breadth-First-Search (BFS)

- Can be considered as a graph traversal




```
BFS (G, s) {
//search for goal
  let Q be Queue
  Q.enqueue(  rootSubtree  )
  mark  rootSubtree  as visited

  while (Q is not empty)
      v = Q.dequeue()
      //process v, is v the goal

     for all neighbors, w, of v in G
        if w is not visited
           Q.enqueue( w )
           mark w as visited
  // easiest way to exit BFS()
}
```

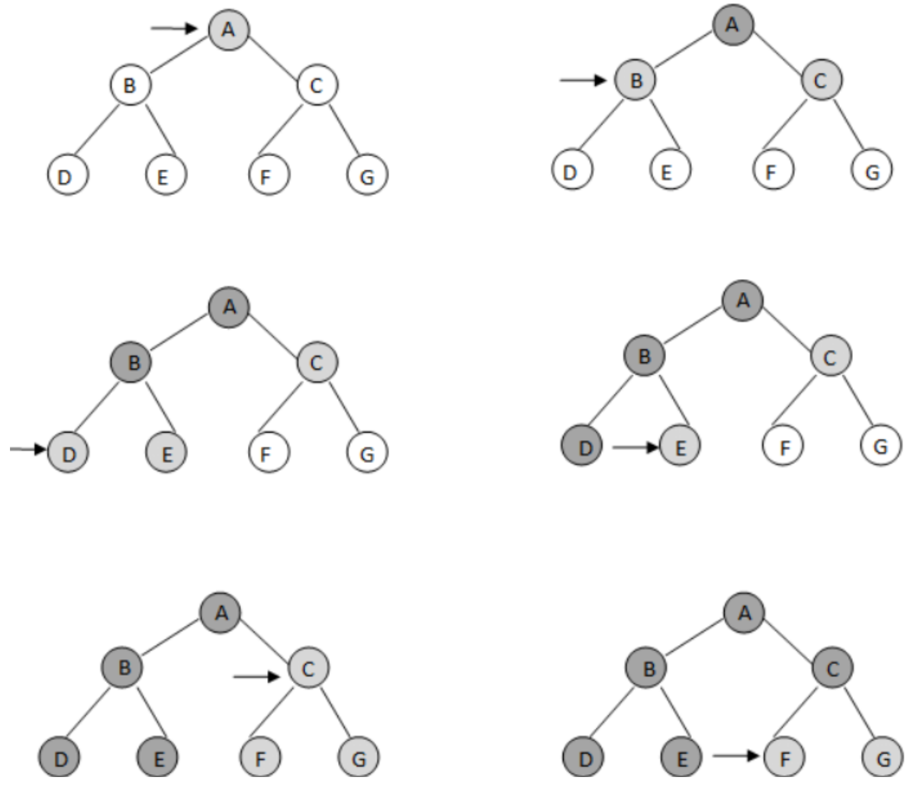
BFS: Expand shallowest first.



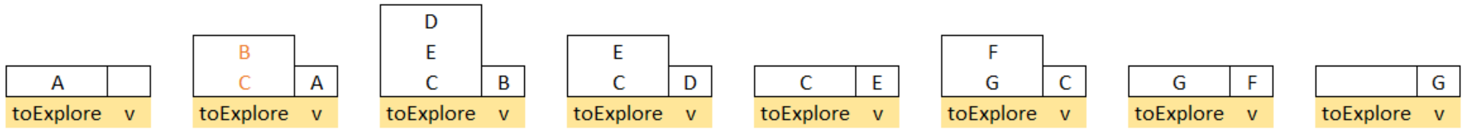
http://www.cs.columbia.edu/~ansaf/courses/4701/AI_campus_search_agents_uninformed.pdf

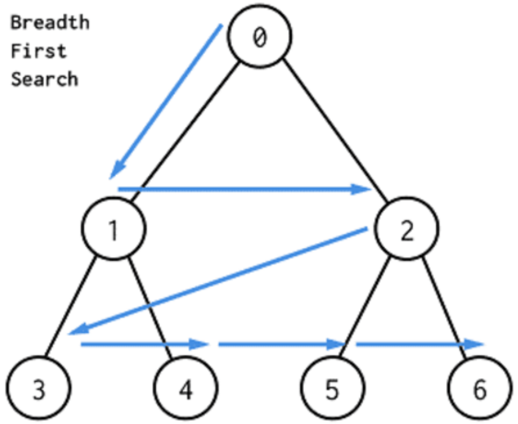
Using Stack to Perform Depth-First-Search (DFS)

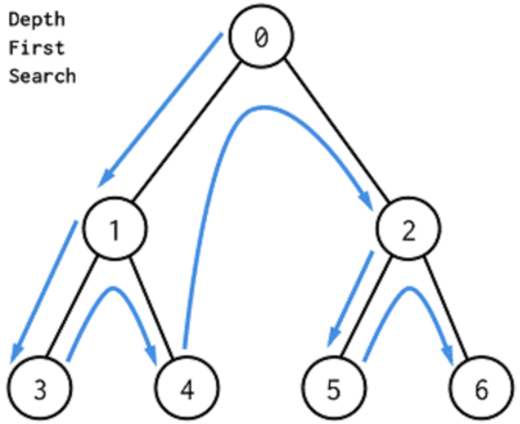
```
      DFS (G, rootSubtree) {
      //search for goal
/*10*/  let toExplore be Stack
/*10*/  toExplore.push( rootSubtree );
/*10*/  mark rootSubtree as visited

/*10*/  while (toExplore is not empty)
/*10*/    v = toExplore.pop()
/*10*/    //if (v is the goal) return v

/*10*/    for all neighbors, w, of v in G
/*10*/      if w is not visited
/*10*/        toExplore.push( w )
/*10*/        mark w as visited

/*10*/  return null; //fail
      }
```



https://www.researchgate.net/figure/Depth-First-Search-progress-251-Depth-First-Search-Algorithm-1-If-the-initial-state-is_fig2_334027256







https://dev.to/danimal92/difference-between-depth-first-search-and-breadth-first-search-6om

src			
1	2		3
	4	5	
6	7	8	dst

					8	dst	
				7	6	6	6
	1	2	4	5	5	5	5
	S	S	S	S	S	S	S
pop		1	2	4	7	8	dst

BFS

```
//   40
//   / \
//  25  78
//  / \
// 10  32


TreeNode root = new TreeNode(x:40);
root.rChild = new TreeNode(x:78);
root.lChild = new TreeNode(x:25);
root.lChild.lChild = new TreeNode(x:10);
root.lChild.rChild = new TreeNode(x:32);
```

```
static void bfs_with_list(TreeNode root) {

  ArrayList<TreeNode> virtual_queue = new ArrayList<>();

  virtual_queue.add(root);

  while (!virtual_queue.isEmpty()) {
    TreeNode n = virtual_queue.remove(0);   // dequeue()


    if (n.lChild != null)
        virtual_queue.add(n.lChild);        // enqueue()
    if (n.rChild != null)
         virtual_queue.add(n.rChild);       // enqueue()
    print(n.data + " "); //40 25 78 10 32
  }
  println();
}
```

```
static void bfs_with_arrayDeque(TreeNode root) {
// avoid using queue interface
  ArrayDeque<TreeNode> queue = new
                               ArrayDeque<>();
  queue.add(root);
  // ArrayDeque add() = append() = enqueue()
  while (!queue.isEmpty()) {
    TreeNode n = queue.remove();
  // ArrayDequeue remove()
  // = remove first element = dequeue()
    if (n.lChild != null)
        queue.add(n.lChild);
    if (n.rChild != null)
        queue.add(n.rChild);
    print(n.data + " "); //40 25 78 10 32
  }
  println();
}
```

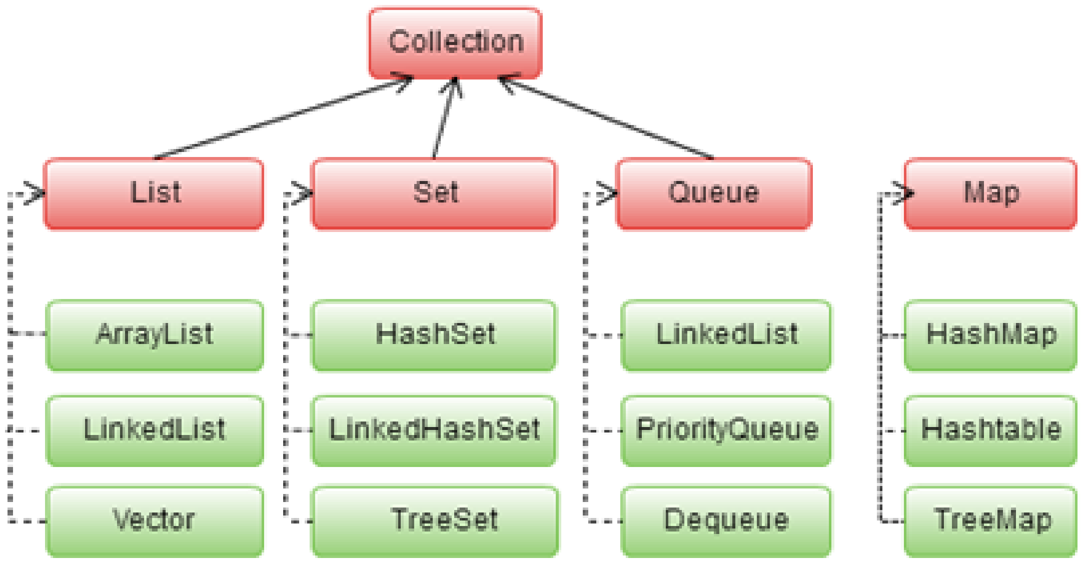
DFS

```java
//   40
//   / \
//  25  78
//  / \
// 10  32


TreeNode root = new TreeNode(x:40);
root.rChild = new TreeNode(x:78);
root.lChild = new TreeNode(x:25);
root.lChild.lChild = new TreeNode(x:10);
root.lChild.rChild = new TreeNode(x:32);
```

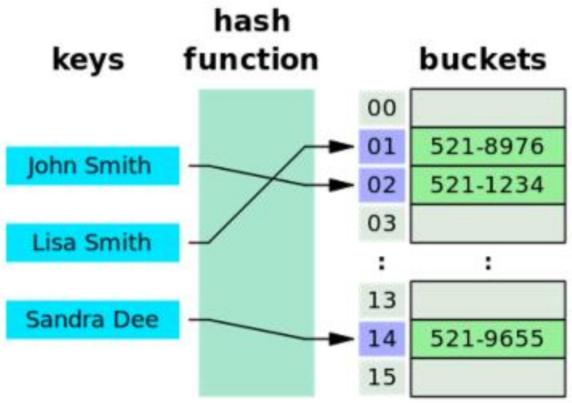
```java
static void dfs_with_list(TreeNode root) {
  ArrayList<TreeNode> virtual_stack = new ArrayList<>();
  virtual_stack.add(0,root);

  while (!virtual_stack.isEmpty()) {
    TreeNode n = virtual_stack.remove(0); // pop()

    if (n.rChild != null)
      virtual_stack.add(0, n.rChild);     // push()
    if (n.lChild != null)
      virtual_stack.add(0, n.lChild);     // push()
    print(n.data + " "); // 40 25 10 32 78
  }
  println();
}
```

```java
static void dfs_with_stack(TreeNode root) {
  Stack<TreeNode> stack = new Stack<>();
  stack.push(root);

  while (!stack.empty()) {
    TreeNode n = stack.pop();

    if (n.rChild != null)
        stack.push(n.rChild);
    if (n.lChild != null)
        stack.push(n.lChild);
    print(n.data + " "); // 40 25 10 32 78
  }
  println();
}
```

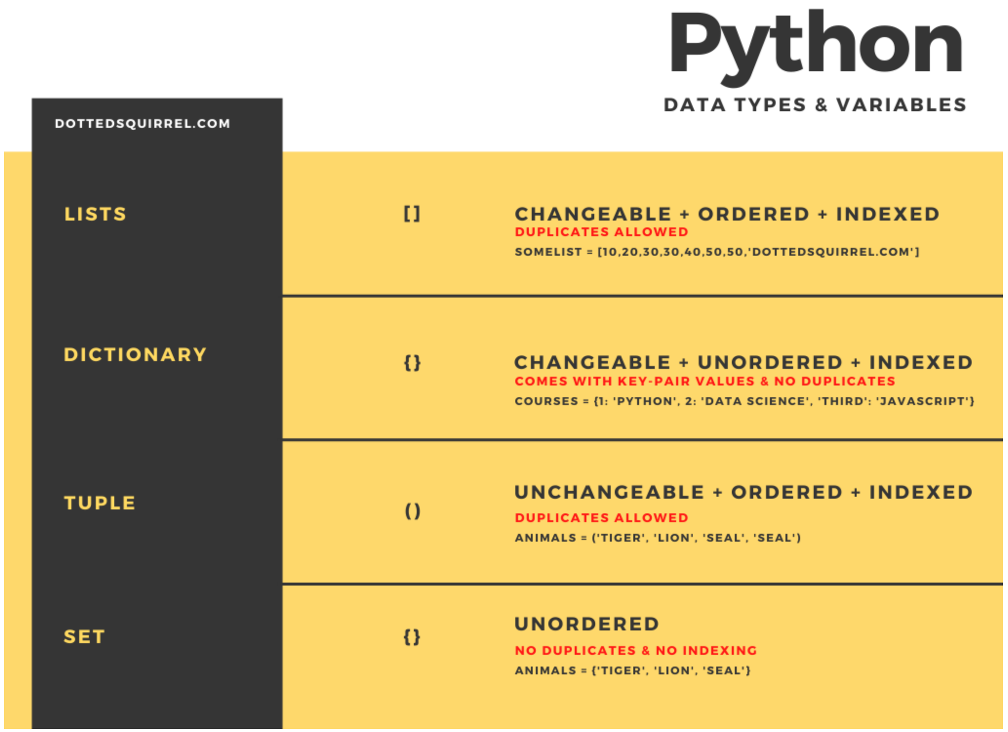
Key ADTs



https://fresh2refresh.com/java-tutorial/java-collections-framework/

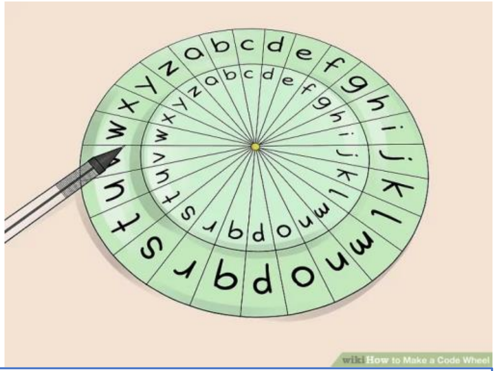


https://en.wikipedia.org/wiki/Hash_table



https://www.dottedsquirrel.com/python-collections/

Caesar Cipher



https://www.wikihow.com/Make-a-Code-Wheel

```
CaesarCipher
This program uses a Caesar cipher for encryption.
Enter encryption key: 3
Plaintext:  ET TU BRUTE
Ciphertext: HW WX EUXWH
```

```
CaesarCipher
This program uses a Caesar cipher for encryption.
Enter encryption key: -3
Plaintext:  HW WX EUXWH
Ciphertext: ET TU BRUTE
```

```java
StringBuffer encrypt(String text, int shift) {
//only captial letters and exclude space_bar

   StringBuffer result = "";
   for (i = 0; i < text.length; i++) {
       char ch = (char)(((int)text.charAt(i) + shift - 65) % 26 + 65);
       result += ch;
   }

   /*decrypt*/ // ch =
   // (char)(((int)text.charAt(i) + (26 - shift) - 65) % 26 + 65);

  return result;
}
```

https://koc.csbridge.org/en/projects/caesarCipher.html

https://www2.seas.gwu.edu/~simhaweb/cs1111/classwork/module14/module14.html

Associative Array & Multidimension Associative Array

Associative Arrays

In an associative array, the keys assigned to values can be arbitrary and user defined strings. In the following example the array uses keys instead of index numbers:

Example — Run this code »

```php
<?php
// Define an associative array
$ages = array("Peter"=>22, "Clark"=>32, "John"=>28);
?>
```

The following example is equivalent to the previous example, but shows a different way of creating associative arrays:

Example — Run this code »

```php
<?php
$ages["Peter"] = "22";
$ages["Clark"] = "32";
$ages["John"] = "28";
?>
```

https://www.tutorialrepublic.com/php-tutorial/php-arrays.php

https://www.geeksforgeeks.org/multidimensional-associative-array-in-php/

```
Array
(
    [Python] => Array
        (
            [first_release] => 1991
            [latest_release] => 3.8.0
            [designed_by] => Guido van Rossum
            [description] => Array
                (
                    [extension] => .py
                    [typing_discipline] => Duck, dynamic, gradual
                    [license] => Python Software Foundation License
                )

        )

    [PHP] => Array
        (
            [first_release] => 1995
            [latest_release] => 7.3.11
            [designed_by] => Rasmus Lerdorf
            [description] => Array
                (
                    [extension] => .php
                    [typing_discipline] => Dynamic, weak
                    [license] => PHP License (most of Zend engine
             under Zend Engine License)
                )
        )

)
```

Summary

- (Data Structure) ADT
- Array vs. List vs. ArrayList
- Early Structure vs. Abstract Data Types
- Queue and Stack
- Tree
- Binary Search Tree
- Tree InOrder Traversal
- Example Stack and Queue Application (BFS, DFS)
- Circular Queue (Array-based)
- HashMap
- Misc

- Other resources
  - https://www.tutorialride.com/data-structures/linked-list-in-data-structure.htm

Linked reference (list of nodes)

```
/* 1 */ public class BookChapter {
/* 2 */    String title;
/* 3 */    int numberOfPages;
/* 4 */    BookChapter next;  // next is a reference of BookChapter Type
/* 5 */    BookChapter(String t, int num) {
/* 6 */        title = t;
/* 7 */        numberOfPages = num;
/* 8 */    }
/* 9 */    public static void main(String[] args) {
/* 10 */        BookChapter anchor, aChapter;
/* 11 */        aChapter = new BookChapter("Prepare", 15);
/* 12 */        anchor= aChapter;
/* 13 */        aChapter = new BookChapter("to follow",25);
/* 14 */        anchor.next = aChapter;
/* 15 */        BookChapter anotherChapter = new BookChapter("the reference",35);
/* 16 */        aChapter.next = anotherChapter;
/* 17 */        System.out.println(anchor.next.next.title); //the reference
/* 18 */        System.out.println(anchor.next.next.numberOfPages); //35
/* 19 */        //another brain exercise
/* 20 */        int totalPageSoFar = anchor.numberOfPages;
/* 21 */        aChapter = anchor.next;
/* 22 */        totalPageSoFar += aChapter.numberOfPages;
/* 23 */        aChapter = aChapter.next;
/* 24 */        totalPageSoFar += aChapter.numberOfPages;
/* 25 */        System.out.println(totalPageSoFar); //75
/* 26 */    }
/* 27 */}
```