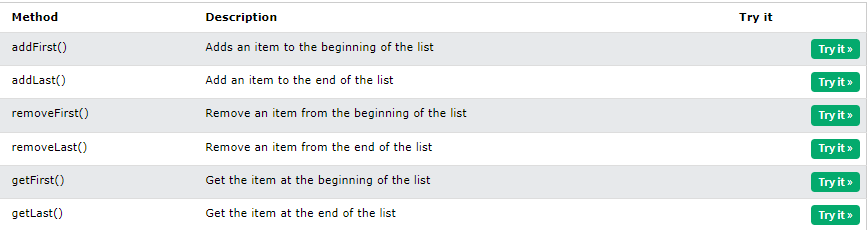
**DATA STRUCTURE AND ALGORITHMS**

1. **ARRAYS**

* Some algorithms of array we can use;

1. Sorting algorithms;
2. Bubble sort;
3. Selection sort;
4. Insertion sort;
5. Search algorithms;
   1. Linear Search
   2. Binary Search
6. **LINKED LISTS**

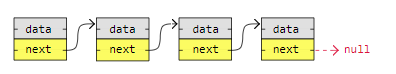
* A list where the nodes are linked together
* Consist of nodes with some sort of data, and a pointer, or link, to the next node.



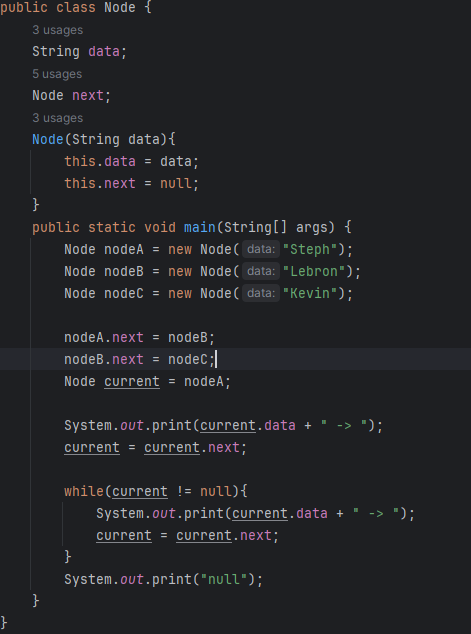
**TYPE OF LINKED LISTS**

1. **SINGLY LINKED LIST**

**-** simplest kind of linked list. It takes up less space in memory because each node has only one address to the next node, like in the image below.

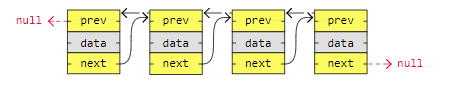


**Example code:**



1. **DOUBLY LINKED LIST**

- has nodes with addresses to both the previous and the next node, like in the image below, and therefore takes up more memory. But doubly linked lists are good if you want to be able to move both up and down in the list.

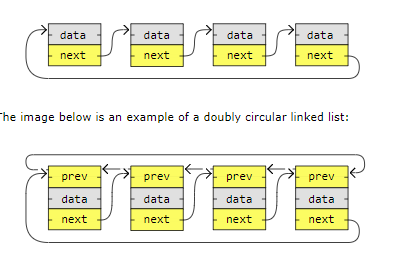


Example code:

public class Nodesss {  
 //doubly linked lists  
 int data;  
 Nodesss next;  
 Nodesss prev;  
 public Nodesss(int data){  
 this.data = data;  
 this.next = null;  
 this.prev = null;  
 }  
 public static void main(String[] args){  
 Nodesss nodeA = new Nodesss(1);  
 Nodesss nodeB = new Nodesss(2);  
 Nodesss nodeC = new Nodesss(3);  
 Nodesss nodeD = new Nodesss(4);  
 //the next  
 nodeA.next = nodeB;  
 nodeB.next = nodeC;  
 nodeC.next = nodeD;  
 //the prev  
 nodeD.prev = nodeC;  
 nodeC.prev = nodeB;  
 nodeB.prev = nodeA;  
  
 System.*out*.println("Forward:");  
 Nodesss current = nodeA;  
 while(current != null){  
 System.*out*.print(current.data + " -> ");  
 current = current.next;  
 }  
 System.*out*.println("\nBackward:");  
 current = nodeD;  
 while(current != null){  
 System.*out*.print(current.data + " -> ");  
 current = current.prev;  
 }  
 }  
}

1. **CIRCULAR LINKED LIST**

**-** is like a singly or doubly linked list with the first node, the "head", and the last node, the "tail", connected.

**-** In singly or doubly linked lists, we can find the start and end of a list by just checking if the links are null. But for circular linked lists, more complex code is needed to explicitly check for start and end nodes in certain applications

.

Example code:

public class Node {  
 String data;  
 Node next;  
 Node(String data){  
 this.data = data;  
 this.next = null;  
 }  
 public static void main(String[] args) {  
 Node nodeA = new Node("Steph");  
 Node nodeB = new Node("Lebron");  
 Node nodeC = new Node("Kevin");  
  
 nodeA.next = nodeB;  
 nodeB.next = nodeC;  
 nodeC.next = nodeA; //this is example of circular linked list  
  
 Node current = nodeA;  
 Node start = nodeA;  
 System.*out*.print(current.data + " -> ");  
 current = current.next;  
  
 while(current != start){  
 System.*out*.print(current.data + " -> ");  
 current = current.next;  
 }  
 System.*out*.print("...");  
 }  
}

Example code with some algorithms:

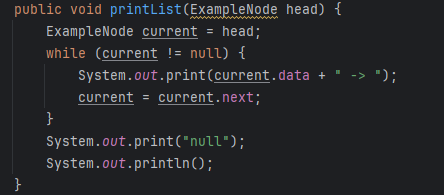
class ExampleNode {  
 int data;  
 ExampleNode next;  
 public ExampleNode(int data){  
 this.data = data;  
 this.next = null;  
 }  
 public String toString(){  
 return String.*valueOf*(data);  
 }

}  
public class Example{  
 static ExampleNode *head*;  
 public void printList(ExampleNode head) {  
 ExampleNode current = head;  
 while (current != null) {  
 System.*out*.print(current.data + " -> ");  
 current = current.next;  
 }  
 System.*out*.print("null");  
 System.*out*.println();  
 }  
 public void sortList() {  
 ExampleNode current = *head*;  
  
 while (current != null) {  
 ExampleNode index = current.next;  
 ExampleNode min = current;  
 // Find the minimum element in the unsorted nodes  
 while (index != null) {  
 if (index.data < min.data) {  
 min = index;  
 }  
 index = index.next;  
 }  
 // Swapping  
 int temp = current.data;  
 current.data = min.data;  
 min.data = temp;  
 // Move to the next node  
 current = current.next;  
 }  
 printList(*head*);  
 }  
 public int lowestNode(ExampleNode head){  
 int value = head.data;  
 ExampleNode current = head;  
 while(current != null){  
 if(value > current.data){  
 value = current.data;  
 }  
 current = current.next;  
 }  
 return value;  
 }  
 public void add(int data){  
 ExampleNode newNode = new ExampleNode(data);  
 if(*head* == null){  
 *head* = newNode;  
 } else {  
 ExampleNode current = *head*;  
 while(current.next != null){  
 current = current.next;  
 }  
 current.next = newNode;  
 }  
 }  
 public void search(ExampleNode head, int target){  
 ExampleNode current = head;  
 int index = 0;  
 while(current != null){  
 if(current.data == target){  
 System.*out*.println(target + " is find in node " + index);  
 return;  
 }  
 current = current.next;  
 index++;  
 }  
 System.*out*.println(target + " is not found in the list..");  
 }  
 public void insertInto(int position, int data){}  
 public static void main(String[] args){  
 Example linkedList = new Example();  
  
 linkedList.add(41);  
 linkedList.add(12);  
 linkedList.add(21);  
 linkedList.add(1);  
  
 linkedList.printList(*head*);  
 System.*out*.println("The Lowest data in the node: " + linkedList.lowestNode(*head*));  
 linkedList.sortList();  
  
 linkedList.search(*head*,21);  
 }  
}

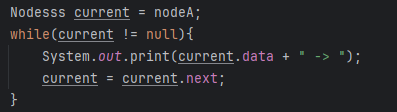
Another example code:

class Nodes {  
 String data;  
 Nodes next;  
 Nodes prev;  
 public Nodes(String data) {  
 this.data = data;  
 this.next = null;  
 this.prev = null;  
 }  
 public String toString(){  
 return String.*valueOf*(data);  
 }  
}  
  
public class LinkedListExample {  
 Nodes head;// this means the first element of the nodes  
 Nodes tail;//last element of the nodes  
 public void add(String data) {  
 Nodes newNode = new Nodes(data);  
 if (head == null) {  
 head = newNode;  
 tail = newNode;  
 } else {  
 tail.next = newNode;  
 newNode.prev = tail;  
 tail = newNode;  
 }  
 }  
 public void printList() {  
 Nodes current = head;  
 System.*out*.println("Forward:");  
 while (current != null) {  
 System.*out*.print(current.data + " -> ");  
 current = current.next;  
 }  
 System.*out*.print("null");  
 System.*out*.println();  
  
 current = tail;  
 System.*out*.println("Backward:");  
 while (current != null) {  
 System.*out*.print(current.data + " -> ");  
 current = current.prev;  
 }  
 System.*out*.print("null");  
 }  
 public void search(Nodes head, String target){  
 Nodes current = head.next;  
 System.*out*.println();  
 while(current != null){  
 if(current.data.toLowerCase().contains(target.toLowerCase())){  
 System.*out*.println(target + " is found at the reference " + current);  
 return;  
 }  
 current = current.next;  
 }  
 System.*out*.println(target + " is not found in list.");  
 }  
  
 public static void main(String[] args) {  
 LinkedListExample list = new LinkedListExample();  
  
 list.add("Micheal Jordan");  
 list.add("Kobe Bryant");  
 list.add("Lebron James");  
 list.add("Steph Curry");  
 list.add("Kevin Durant");  
  
 list.printList();  
 list.search(list.head, "lebron");  
 }  
}

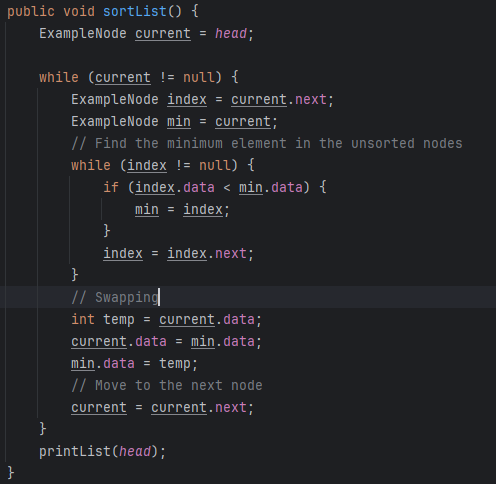
To print the nodes with a method and its parameter;



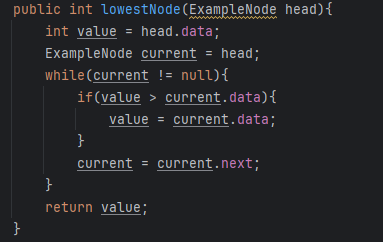
In the main method;



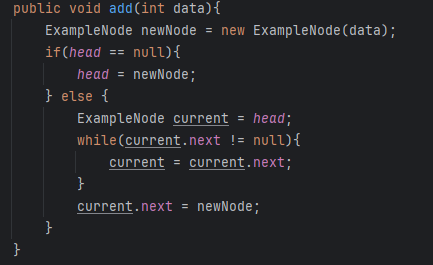
To sort the nodes with a method;



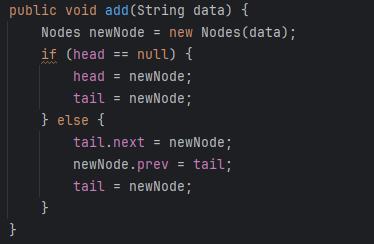
Find the lowest Node;



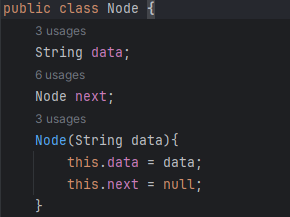
To add a data into nodes with singly linked list;



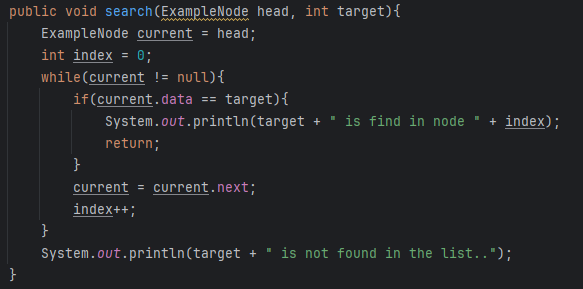
To add a data into nodes with doubly linked list;



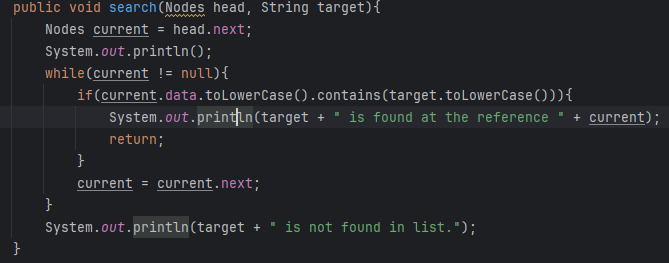
The basic syntax and important attributes in node;



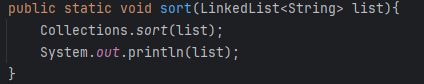
Searching integer method with its parameter target;



Searching string method with its parameter target;



To sort a String;



**3. STACKS**

- Data structure that can hold many elements.

- LIFO data structure – **Last-in First-out.**

- stores object into sort of vertical tower.

- think of an cd na ga patong patong, means if mo delete ta that means ag naa sa babaw maoy ma delete una, ug mo add sad ta then mo patong siya if delete ta ag bag o na ma add mao ag ma delete. And we have methods to delete to add an stacks.

Used methods in stacks:

1. **push()** – to add
2. **pop()** – to remove and returns the top element
3. **peek() –** returns the top element
4. **isEmpty()** – check if stack is empty
5. **size() –** find or returns the number of elements
6. **search()** – it returns the number of element where is the target located and if its not in the stacks then it returns -1

- Stacks can be implemented by arrays or linked lists.

- Stacks can be used to implement undo mechanisms, to revert to previous states, to create algorithms for depth-first search in graphs, or for backtracking.

- Stacks are often mentioned together with Queues, which is a similar data structure

**Reasons why implement arrays:**

- **Memory Efficient:**Array elements do not hold the next elements address like linked list nodes do.

**- Easier to implement and understand:**Using arrays to implement stacks require less code than using linked lists, and for this reason it is typically easier to understand as well.

**Reasons why implement linked lists:**

**- Dynamic size:**The stack can grow and shrink dynamically, unlike with arrays.

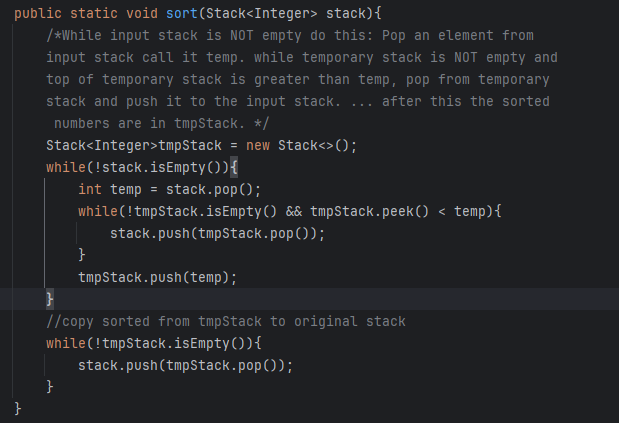
Example code:

import java.util.Stack;  
public class Main {  
 public static void sort(Stack<Integer> stack){  
 /\*While input stack is NOT empty do this: Pop an element from  
 input stack call it temp. while temporary stack is NOT empty and  
 top of temporary stack is greater than temp, pop from temporary  
 stack and push it to the input stack. ... after this the sorted  
 numbers are in tmpStack. \*/  
 Stack<Integer>tmpStack = new Stack<>();  
 while(!stack.isEmpty()){  
 int temp = stack.pop();  
 while(!tmpStack.isEmpty() && tmpStack.peek() < temp){  
 stack.push(tmpStack.pop());  
 }  
 tmpStack.push(temp);  
 }  
 //copy sorted from tmpStack to original stack  
 while(!tmpStack.isEmpty()){  
 stack.push(tmpStack.pop());  
 }  
 }  
 public static void search(Stack<Integer> stack,int target){  
 int position = stack.search(target);  
  
 if(position != -1){  
 System.*out*.println(target + " is found at element " + position);  
 }  
 if(position == -1){  
 System.*out*.println(target + " is not found in stack");  
 }  
 }  
 public static void main(String[] args) {  
 Stack<Integer> stack = new Stack<>();  
  
 stack.push(4);  
 stack.push(2);  
 stack.push(6);  
 stack.push(12);  
 stack.push(1);  
  
 System.*out*.println("Before sorting:");  
 System.*out*.println(stack);  
 //stack.pop();  
 //System.out.println(stack);  
 System.*out*.println(stack.search(2));  
 *search*(stack, 12);  
  
 *sort*(stack);  
 System.*out*.println("After sorting:");  
 System.*out*.println(stack);  
 //System.out.println(stack.peek());  
 System.*out*.println(stack.search(4));  
 *search*(stack, 12);  
 }  
}

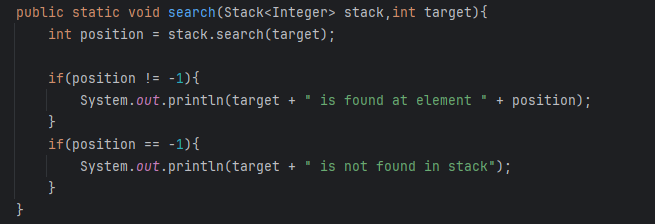
Another example code:

import java.util.Stack;  
import java.util.Collections;  
import java.util.ArrayList;  
  
public class StringStacks {  
 public static void sort(Stack<String> stack){  
 ArrayList<String> list = new ArrayList<>(stack);  
 Collections.*sort*(list);  
 stack.clear();  
  
 for(String word : list){  
 stack.push(word);  
 }  
 System.*out*.println(stack);  
 }  
 public static void search(Stack<String> stack, String target){  
 int position = stack.search(target);  
  
 if(position != -1){  
 System.*out*.println(target + " is found in element " + position);  
 } else{  
 System.*out*.println(target + " is not found in stack");  
 }  
 }  
 public static void main(String[] args){  
 Stack<String> bini = new Stack<>();  
  
 bini.push("Colet");  
 bini.push("Mikha");  
 bini.push("Aiah");  
 bini.push("Stacey");  
 bini.push("Jhoanna");  
 bini.push("Maloi");  
 bini.push("Gwen");  
 bini.push("Sheena");  
  
 System.*out*.println("Before sorting:");  
 System.*out*.println(bini);  
 System.*out*.println(bini.peek()); // output: Sheena  
 *search*(bini, "Mikha");  
  
 System.*out*.println("After sorting:");  
 *sort*(bini);  
 System.*out*.println(bini.peek()); // output: Stacey  
 *search*(bini, "Mikha");  
 }  
}

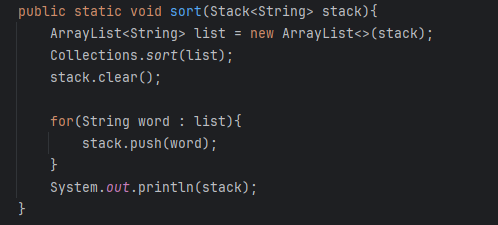
To sort an integer;



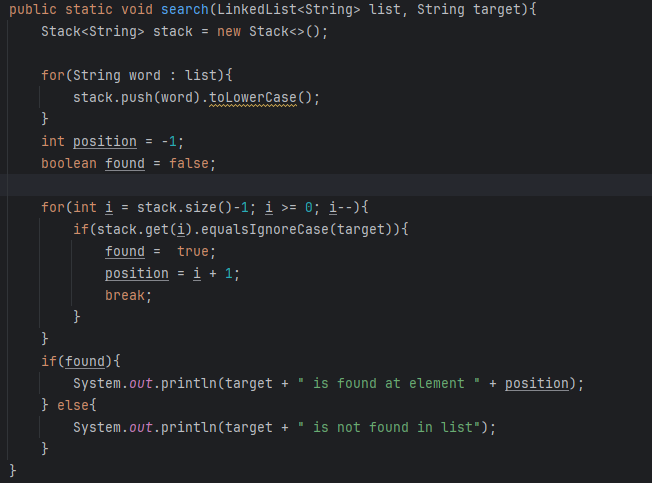
To search an integer;



To sort a String;



To search a String with ignore case from linked list and change the first parameter to use an stack;



**4. Queues**

- a **FIFO(**First-In First-Out**)** – think of an people in line.

- a data structure that can hold many elements.

- a collection designed for holding elements prior to processing Linear Data Structure.

- queues can be implement in arrays or linked list.

Some methods or operations used:

1. **Enqueue** – adds a new element. = **offer();**
2. **Dequeue** – remove and return the front element. = **poll();**
3. **Peek** – return the front or first element.
4. **isEmpty** – to check if the queue is empty.
5. **Size** – return the number of elements.

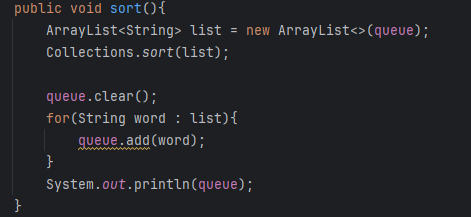
Example code with basic syntax using built-in libraries;

import java.util.LinkedList;  
import java.util.Queue;  
  
public class Main {  
 public static void main(String[] args) {  
 Queue<String> queue = new LinkedList<>();  
 // queue is not a class, it is an interface that is why we used linkedlist  
  
 queue.offer("Paul");// offer() to add  
 queue.offer("Chad");   
 queue.offer("Bro");  
 queue.offer("Xam");  
 System.*out*.println(queue);  
 System.*out*.println(queue.peek()); // peek() for peeking the first element  
 }  
}

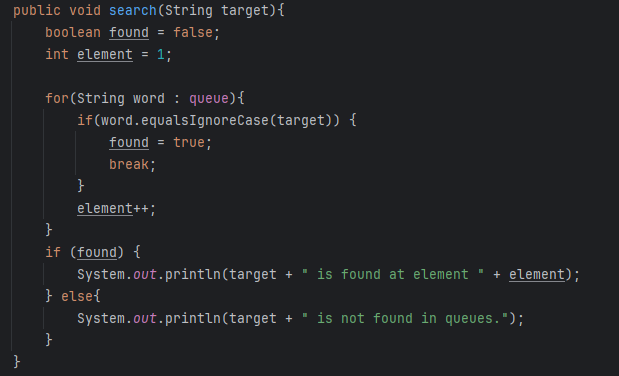
Example code with sorting and linear search algo with implementing enqueue and dequeue using built-in library, Strings;

import java.util.Queue;  
import java.util.LinkedList;  
import java.util.ArrayList;  
import java.util.Collections;  
  
public class Queues {  
 String data;  
 int front;  
 int size;  
 int capacity;  
 Queue<String> queue = new LinkedList<>();  
 public Queues(int capacity) {  
 this.capacity = capacity;  
 this.data = new String();  
 this.front = 0;  
 this.size = 0;  
 }  
 public void enqueue(String data){  
 queue.offer(data);  
 }  
 public void dequeue(){  
 queue.poll();  
 }  
 public void printQueue(){  
 System.*out*.println(queue);  
 }  
 public String peek(){  
 return queue.peek();  
 }  
 public boolean isEmpty(){  
 return queue.isEmpty();  
 }  
 public int size(){  
 return queue.size();  
 }  
 public void sort(){  
 ArrayList<String> list = new ArrayList<>(queue);  
 Collections.*sort*(list);  
  
 queue.clear();  
 for(String word : list){  
 queue.add(word);  
 }  
 System.*out*.println(queue);  
 }  
 public void search(String target){  
 boolean found = false;  
 int element = 1;  
  
 for(String word : queue){  
 if(word.equalsIgnoreCase(target)) {  
 found = true;  
 break;  
 }  
 element++;  
 }  
 if (found) {  
 System.*out*.println(target + " is found at element " + element);  
 } else{  
 System.*out*.println(target + " is not found in queues.");  
 }  
 }  
}  
class Mains{  
 public static void main(String[] args){  
 Queues bini = new Queues(10);  
  
 bini.enqueue("Colet");  
 bini.enqueue("Mikha");  
 bini.enqueue("Aiah");  
 bini.enqueue("Stacey");  
 bini.enqueue("Jhoanna");  
 bini.enqueue("Maloi");  
 bini.enqueue("Gwen");  
 bini.enqueue("Sheena");  
  
 System.*out*.println("Before sorting:");  
 bini.printQueue();  
 System.*out*.println("peek: " + bini.peek());  
 //bini.dequeue();  
 //bini.printQueue();  
 bini.search("Mikha");  
  
 System.*out*.println();  
 System.*out*.println("After sorting:");  
 bini.sort();  
 System.*out*.println("peek: " + bini.peek());  
 //bini.dequeue();  
 //bini.printQueue();  
 bini.search("Mikha");  
  
 }  
}

Sorting algorithm with string;



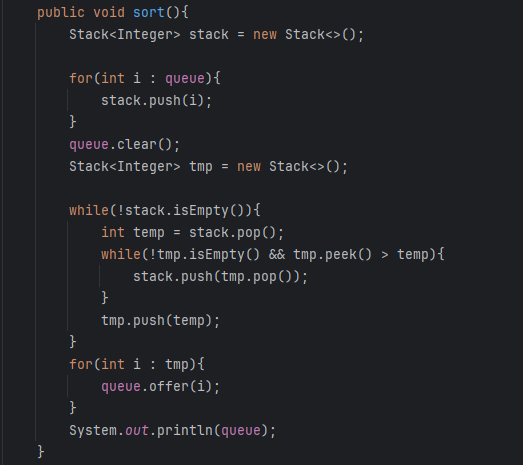
Search algorithm with string;



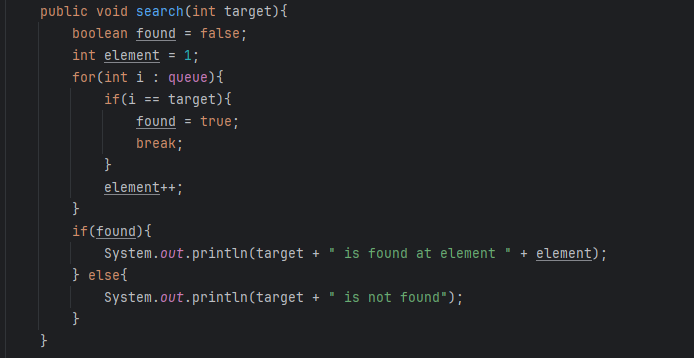
Example code with sorting and linear search algo with implementing enqueue and dequeue using built-in library, Integer;

import java.util.LinkedList;  
import java.util.Queue;  
import java.util.Stack;  
  
public class NumberQueue {  
 int[] data;  
 int capacity;  
 int size;  
 Queue<Integer> queue = new LinkedList<>();  
 public NumberQueue(int capacity){  
 this.capacity = capacity;  
 this.data = new int[capacity];  
 this.size = 0;  
 }  
 public void enqueue(int data){  
 queue.offer(data);  
 size++;  
 }  
 public void dequeue(){  
 queue.poll();  
 }  
 public boolean isEmpty(){  
 return queue.isEmpty();  
 }  
 public int size(){  
 this.size = queue.size();  
 return queue.size();  
 }  
 public boolean isFull(){  
 return size == capacity;  
 }  
 public int peek(){  
 return queue.peek();  
 }  
 public void sort(){  
 Stack<Integer> stack = new Stack<>();  
  
 for(int i : queue){  
 stack.push(i);  
 }  
 queue.clear();  
 Stack<Integer> tmp = new Stack<>();  
  
 while(!stack.isEmpty()){  
 int temp = stack.pop();  
 while(!tmp.isEmpty() && tmp.peek() > temp){  
 stack.push(tmp.pop());  
 }  
 tmp.push(temp);  
 }  
 for(int i : tmp){  
 queue.offer(i);  
 }  
 System.*out*.println(queue);  
 }  
 public void search(int target){  
 boolean found = false;  
 int element = 1;  
 for(int i : queue){  
 if(i == target){  
 found = true;  
 break;  
 }  
 element++;  
 }  
 if(found){  
 System.*out*.println(target + " is found at element " + element);  
 } else{  
 System.*out*.println(target + " is not found");  
 }  
 }  
 public void printQueue(){  
 System.*out*.println(queue);  
 }  
}  
class Mainss{  
 public static void main(String[] args){  
 NumberQueue num = new NumberQueue(10);  
  
 num.enqueue(5);  
 num.enqueue(1);  
 num.enqueue(3);  
 num.enqueue(9);  
 num.enqueue(7);  
  
 System.*out*.println("Before sorting:");  
 num.printQueue();  
 System.*out*.println("peek: " + num.peek());  
 System.*out*.println(num.isFull());  
 num.search(3);  
  
 System.*out*.println();  
 System.*out*.println("After sorting:");  
 num.sort();  
 System.*out*.println("peek: " + num.peek());  
 num.search(3);  
 }  
}

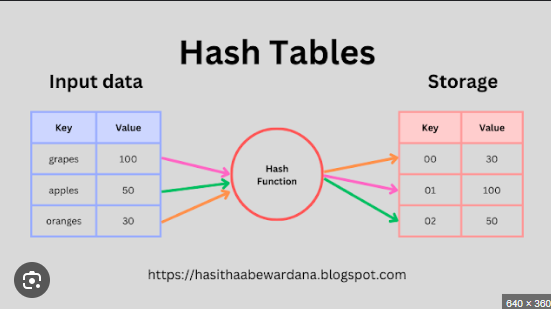
Sorting algorithm in dealing numbers;

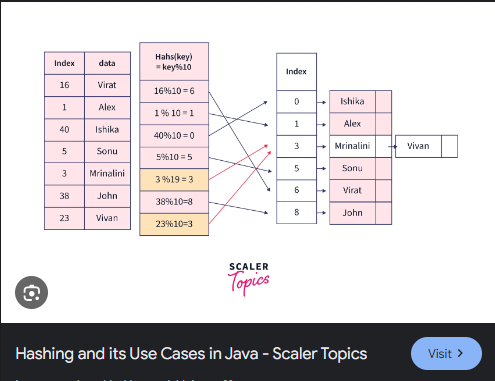


Linear search algorithm in dealing numbers;



**4. Hash Table**

****

****

**-** A Hash Table is a data structure designed to be fast to work with.

- The reason Hash Tables are sometimes preferred instead of arrays or linked lists is because searching for, adding, and deleting data can be done really quickly, even for large amounts of data.

- A data structure stores unique keys to values ex. <Integer, String>

- hashtable is not inherently suitable for sorting because it does not maintain any order of its elements.

- Each key/value pair is known as Entry

- FAST insertion, look up, deletion of key/value pairs

- Not ideal for small data sets, great with large data sets

**Hashing –** takes a key and computes an integer (formula will vary based on key & data types)

- in hashtable, we use the hash % capacity to calculate the index number ex. Key.hashCode % capacity = index

**Bucket –** an indexed storage location for one or more Entries

- can store multiple Ent ries in case of a collision (linked similarly a LinkedList)

**Collision –** hash function generates the same index for more than one key

- less collision = more efficiency

**METHODS used in HASHTABLE**

1. **clear()**: Removes all key-value mappings from the hashtable.
2. **clone()**: Creates a shallow copy of the hashtable.
3. **contains(Object value)**: Checks if the hashtable contains a value.
4. **boolean containsKey(Object key)**: Checks if the hashtable contains a specific key.
5. **boolean containsValue(Object value)**: Checks if the hashtable contains a specific value.
6. **Enumeration elements()**: Returns an enumeration of the values in the hashtable.
7. **Set<Map.Entry<K, V>> entrySet()**: Returns a set view of the mappings contained in the hashtable.
8. **boolean equals(Object o)**: Compares the specified object with the hash table for equality.
9. **V get(Object key)**: Returns the value to which the specified key is mapped, or null if the hashtable contains no mapping for the key.
10. **int hashCode()**: Returns the hash code value for the hashtable.
11. **boolean isEmpty()**: Checks if the hashtable is empty.
12. **Enumeration<K> keys()**: Returns an enumeration of the keys in the hashtable.
13. **Set<K> keySet()**: Returns a set view of the keys contained in the hashtable.
14. **V put(K key, V value)**: Maps the specified key to the specified value in the hashtable.
15. **void putAll(Map<? extends K, ? extends V> t)**: Copies all of the mappings from the specified map to the hashtable.
16. **V remove(Object key)**: Removes the key (and its corresponding value) from the hashtable.
17. **int size()**: Returns the number of key-value mappings in the hashtable.
18. **Collection<V> values()**: Returns a collection view of the values contained in the hashtable.

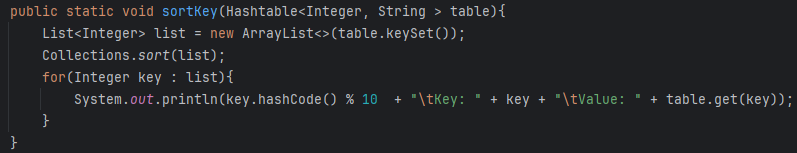
Example code:

import java.util.\*;  
  
public class Main {  
 public static void sortKey(Hashtable<Integer, String > table){  
 List<Integer> list = new ArrayList<>(table.keySet());  
 Collections.*sort*(list);  
 for(Integer key : list){  
 System.*out*.println(key.hashCode() % 10 + "\tKey: " + key + "\tValue: " + table.get(key));  
 }  
 }  
 public static void sortValue(Hashtable<Integer, String> hashtable){  
 List<Map.Entry<Integer,String>> list = new ArrayList<>((Collection) hashtable.entrySet());  
 Collections.*sort*(list, (e1, e2)->e1.getValue().compareTo(e2.getValue()));  
  
 for(Map.Entry<Integer, String> key : list){  
 System.*out*.println(key.getKey() % 10 + "\tKey: " + key.getKey() + "\tValue: " + key.getValue());  
 }  
 }

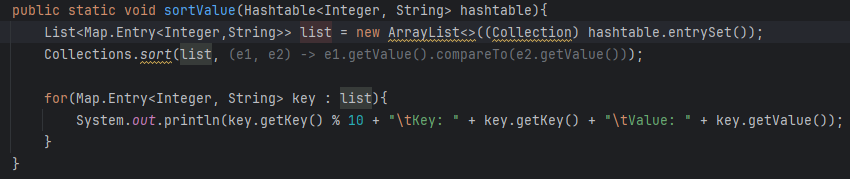
public static void search(Hashtable<Integer, String> table, Object target){  
 boolean found = false;  
  
 for (Map.Entry<Integer, String> entry : table.entrySet()) {  
 if (entry.getValue().equals(target)) {  
 System.*out*.println("Value \"" + target + "\" is found in the table.");  
 System.*out*.println("\t" + entry.getKey() % 10 + "\tKey: " + entry.getKey() + "\tValue: " + entry.getValue());  
 found = true;  
 } else if(entry.getKey().equals(target)){  
 System.*out*.println("Value \"" + target + "\" is found in the table.");  
 System.*out*.println("\t" + entry.getKey() % 10 + "\tKey: " + entry.getKey() + "\tValue: " + entry.getValue());  
 found = true;  
 }  
 }  
 if (!found) {  
 System.*out*.println("Value \"" + target + "\" not found in the table.");  
 }  
 }

public static void main(String[] args) {  
 Hashtable<Integer, String> bini = new Hashtable<>(10);  
  
 bini.put(100, "Mikha");  
 bini.put(124, "Colet");  
 bini.put(123, "Aiah");  
 bini.put(871, "Stacy");  
 bini.put(915, "Maloi");  
 bini.put(787, "Sheena");  
 bini.put(516, "Gwen");  
 bini.put(122, "Jhoanna");  
  
  
 //bini.remove(122);  
  
 System.*out*.println("Before sorting:");  
 for(Integer key : bini.keySet()){  
 System.*out*.println(key.hashCode() % 10 + "\tKey: " + key.hashCode() + "\tValue: " + bini.get(key));  
 }  
  
 System.*out*.println();  
 System.*out*.println("After sorting with the key:");  
 *sortKey*(bini);  
  
 System.*out*.println();  
 System.*out*.println("After sorting with the value:");  
 *sortValue*(bini);  
  
 System.*out*.println();  
 *search*(bini, "Mikha");  
 *search*(bini, "Mikhhaaa");  
 }  
}

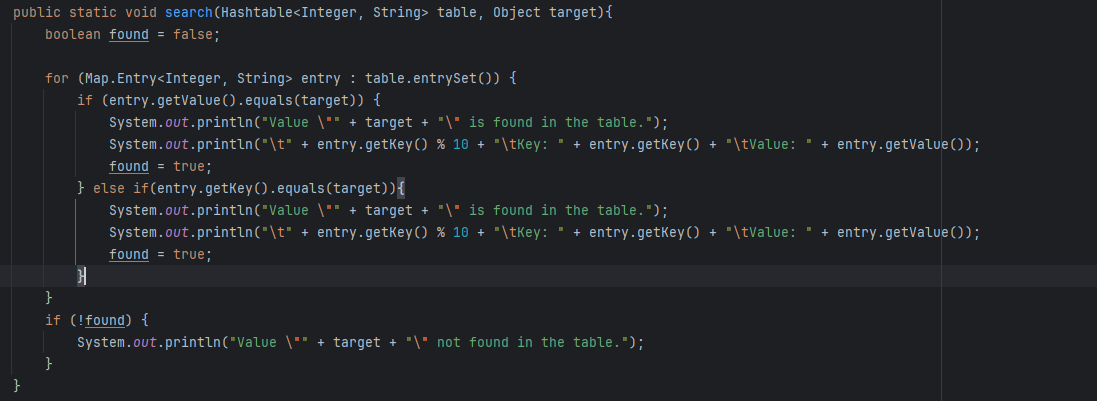
hashtable is not suitable for sorting coz of the key/value pair, but here some code to **sort an hashtable by their key(integer)**;



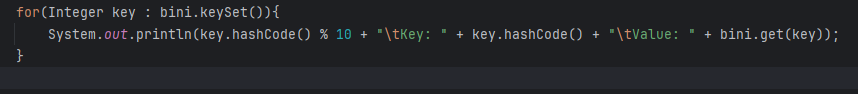
Sort an hashtable by their value(string);



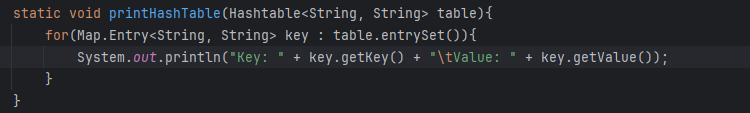
Search the value or key algorithms in hashtable;



To print with Integer key;



To print with String key;



Another example code;

import java.util.\*;  
public class InteHashT {  
 static void print(Hashtable<String, Integer> table){  
 for(String key : table.keySet()){  
 System.*out*.println("Key: " + key + "\tValue: " + table.get(key));  
 }  
 }  
 static void sortKey(Hashtable<String, Integer> table){  
 List<String> list = new ArrayList<>(table.keySet());  
 Collections.*sort*(list);  
  
 for(String key : list){  
 System.*out*.println("Key: " + key + "\tValue: " + table.get(key));  
 }  
 }  
 static void sortValue(Hashtable<String, Integer> table){  
 List<Map.Entry<String, Integer>> entries = new ArrayList<>(table.entrySet());  
 Collections.*sort*(entries, (e1, e2) -> e1.getValue().compareTo(e2.getValue()));  
  
 for(Map.Entry<String, Integer> entry : entries){  
 System.*out*.println("Key: " + entry.getKey() + "\tValue: " + entry.getValue());  
 }  
 }  
 static void search(Hashtable<String, Integer> table, Object target){  
 boolean found = false;  
  
 for(String key : table.keySet()){  
 if(key == target){  
 System.*out*.println("Key " + target + " is found in table.");  
 System.*out*.println("\tKey: " + key + "\tValue: " + table.get(key));  
 found = true;  
 } else if(table.get(key) == target){  
 System.*out*.println("Value " + target + " is found in table.");  
 System.*out*.println("\tKey: " + key + "\tValue: " + table.get(key));  
 found = true;  
 }  
 }  
 if(!found){  
 System.*out*.println(target + " is not found in table");  
 }  
 }  
  
 public static void main(String[] args){  
 Hashtable<String, Integer> num = new Hashtable<>();  
  
 num.put("A", 1);  
 num.put("C", 3);  
 num.put("B", 2);  
  
 System.*out*.println("Before sorting: ");  
 *print*(num);  
  
 System.*out*.println();  
 System.*out*.println("After sort with a key:");  
 *sortKey*(num);  
  
 System.*out*.println();  
 System.*out*.println("After sort with a value:");  
 *sortValue*(num);  
  
 System.*out*.println();  
 System.*out*.println("Searching:");  
 *search*(num,1);  
 *search*(num, "B");  
 *search*(num, 5);  
 }  
}

**6. HASH SET**

- hash set is a form of hash table data structure that usually holds a large number of elements.

- Using a Hash Set we can search, add, and remove elements really fast.

- Hash Sets are used for lookup, to check if an element is part of a set.