- Tree VS List
- Vector VS ArrayList
- Array VS ArrayList
- ArrayList VS LinkedList
- Linked List vs Array
- HashTable
- Heap VS BinaryTree
- BST VS HashTable
- HashTable VS HashMap
- External Sort VS K- way merge
- Merge Sort vs Quick Sort
- Design Pattern
- Difference between Interface & Abstract class
- Overload VS Override
- Grep VS Find
- Logical vs. Bitwise Operators
- DFS VS BFS

- Primitive Type
- Java static method
- Abstract class and Interface
- Object VS Class
- final , finally , finalize()
- Polymorphism and encapsulation
- Inheritance vs Composition
- Inheritance Advantages and Disadvantages:
- Process VS Thread
- synchronized VS volatile
- TreeMap

Tree VS List

In a linked list, the items are linked together through a single next pointer. In a binary tree, each node can have 0, 1 or 2 subnodes, where (in case of a binary search tree) the key of the left node is lesser than the key of the node and the key of the right node is more than the node. As long as the tree is balanced, the search path to each item is a lot shorter than that in a linked list.

- Binary Search Trees
  - medium complexity to implement (assuming you can't get them from a library)
  - inserts are O(logN)

    - lookups are O(logN)
- Linked lists (unsorted)
    - low complexity to implement
    - inserts are O(1)
    - lookups are O(N)

Vector VS ArrayList

Vector is almost identical to ArrayList, and the difference is that Vector is synchronized. Because of this, it has an overhead than ArrayList. Normally, most Java programmers use ArrayList instead of Vector because they can synchronize explicitly by themselves.

External Sort And K- way merge

The first step is quite easy: we open a stream reader on the big file and keep reading until we have enough data in our buffer to fill a file chunk. When the buffer is full, we sort the data in memory and write it on disk on a temp file. We continue until the whole input file is processed. As a result, we have K chunk files on disk.

Now that we have K small files filled with sorted data, we are going to merge them in order to build our final result file. To perform the merge, we need first to open K file streams, one for each chunk file. Then, we have to read one line per file and take the smallest line each time. We repeat this operation until we finish reading all the chunks. In order to avoid doing a lot of string comparisons at each iteration, we keep the current line values in an ordered structure.

The algorithm(Two – way sort) requires [log(N/M)] passes plus the initial run-constructing pass. At each pass we process N records, so the complexity isO(Nlog(N/M))

K – way merge: O(Nlogk(N/M))

N records on Ta1

M records can fit in the memory

Design Pattern

Design patterns represent the best practices used by experienced object-oriented software developers. Design patterns are solutions to general problems that software developers faced during software development. These solutions were obtained by trial and error by numerous software developers over quite a substantial period of time.

Design patterns provide a standard terminology and are specific to particular scenario. For example, a singleton design pattern signifies use of single object so all developers familiar with single design pattern will make use of single object and they can tell each other that program is following a singleton pattern.

Factory Pattern: Creates objects without exposing the instantiation logic to the client and Refers to the newly created object through a common interface.

http://www.tutorialspoint.com/design_pattern/factory_pattern.htm

Singleton Pattern: This pattern involves a single class which is responsible to create an object while making sure that only single object gets created. This class provides a way to access its only object

which can be accessed directly without need to instantiate the object of the class.

http://www.tutorialspoint.com/design_pattern/singleton_pattern.htm

MVC

Difference between Interface & Abstract class

	abstract Classes	Interfaces
1	abstract class can extend only one class or one abstract class at a time	interface can extend any number of interfaces at a time
2	abstract class can extend from a class or from an abstract class	interface can extend only from an interface
3	abstract class can have both abstract and concrete methods	interface can have only abstract methods
4	A class can extend only one abstract class	A class can implement any number of interfaces
5	In abstract class keyword 'abstract' is mandatory to declare a method as an abstract	In an interface keyword 'abstract' is optional to declare a method as an abstract
6	abstract class can have protected , public and public abstract methods	Interface can have only public abstract methods i.e. by default
7	abstract class can have static, final or static final variable with any access specifier	interface can have only static final (constant) variable i.e. by default

When to use Abstract class and interface:

- If you have a lot of methods and want default implementation for some of them, then go with abstract class

- If you want to implement multiple inheritance then you have to use interface.
- If your base contract keeps on changing, then you should use abstract class, as if you keep changing your base contract and use interface, then you have to change all the classes which implements that interface.

Merge Sort vs Quick Sort :

Merge Sort: O(nlog(n))

Quick Sort: ave O(nlog(n)), worst O(n^2)

Quick sort is in-place sorting algorithm while the merge sort is not in-place. In merge sort, to merge the sorted arrays it requires a temporary array and hence it is not in-place.

However time efficiency of the quick sort depends on the choice of the pivot element. Normally the middle or median element is chosen.

Stability: merge sort: stable; quick sort: not stable
Best fit: merge sort: slow sequential media; quick sort: general purpose.
Merge sort is very efficient for immutable data structures like linked lists and is therefore a good choice for (purely) functional programming languages.

HashTable 123:

Diff of HashMap, HashTable, HashSet. Only HashTable is thread safe.

A hash table is made up of two parts: an array (the actual table where the data to be searched is stored) and a mapping function, known as a hash function. The hash function is a mapping from the input space to the integer space that defines the indices of the array. In other words, the hash function provides a way for assigning numbers to the input data such that the data can then be stored at the array index corresponding to the assigned number.

A hash function doesn't guarantee that every input will map to a different output. There is always the chance that two inputs will hash to the same output. This indicates that both elements should be inserted at the same place in the array, and this is impossible. This phenomenon is known as a collision.

There are many algorithms for dealing with collisions, such as linear probing(抢占别人位置，不能 delete) and separate chaining（存 linkedlist）. Separate chaining requires a slight modification to the data structure. Instead of storing the data elements right into the array, they are stored in linked lists. Each slot in the array then points to one of these linked lists. When an element hashes to a value, it is added to the linked list at that index in the array. Because a linked list has no limit on length, collisions are no longer a problem. If more than one element hashes to the same value, then both are stored in that linked list.

Separate chaining allows us to solve the problem of collision in a simple yet powerful manner. Of course, there are some drawbacks. **Imagine the worst case scenario where through some fluke of bad luck and bad programming, every data element hashed to the same value.** In that case, to do a lookup, we'd really be doing a straight linear search on a linked list, which means that our search operation is back to being $O(n)$. The worst case search time for a hash table is$O(n)$. However, the probability of that happening is so small that, while the worst case search time is $O(n)$, both the best and average cases are $O(1)$

Search, insert, delete O(1)

Collision: Open hashing 存的是 ListNode LinkedList

Closed hashing 位置被占，找别的位置占。不支持删除！不

Solution： rehashing. But the current thread will be locked.

Array VS ArrayList

ARRAY	ARRAYLIST
Stores primitive data types and also objects	Stores only objects
Defined in Java language itself as a fundamental data structure	Belongs to collections framework
Fixed size	Growable and resizable. Elements can be added or removed
Stores similar data of one type	Can store heterogeneous data types
It is not a class	It is a class with many methods
Cannot be synchronized	Can be obtained a synchronized version
Elements retrieved with for loop	Can be retrieved with for loop and iterators
Elements accessible with index number	Accessing methods like get() etc. are available
Can be multidimensional	—
Set, Check element at a particular index: **O(1); Searching**: **O(n)** if array is unsorted and **O(log n)** if array is sorted and something like a binary search is used	• **Add**: **O(1) – O(n)** • **Remove**: O(1) - **O(n)** • **Contains**: **O(n)** • **Size**: **O(1)**

ArrayList VS LinkedList

There comes two classes ArrayList and LinkedList to store objects. Internally, ArrayList stores elements as an array form and LinkedList stores elements in node form. Due to their internal style of storing the elements, they come with different performance heads depending the nature of action performed like addition and retrieval.

1) **Search**: ArrayList : O(1) LinkedList: O(n)
2) **Deletion**: LinkedList remove operation gives `O(1)` performance
   ArrayList gives variable performance: `O(n)` in worst case (while removing first element) and `O(1)` in best case (While removing last element).

Conclusion: LinkedList element deletion is faster compared to ArrayList.

3) **Inserts Performance**: LinkedList add method gives `O(1)` performance while ArrayList gives `O(n)` in worst case.

4) **Memory Overhead**: ArrayList maintains indexes and element data while LinkedList maintains element data and two pointers for neighbor nodes hence the memory consumption is high in LinkedList comparatively.

	LinkedList	ArrayList
Adding Element	Cheap	Expensive
Retrieving Element	Expensive	Cheap

When to use LinkedList and when to use ArrayList?

1) As explained above the insert and remove operations give good performance (`O(1)`) in LinkedList compared to ArrayList(`O(n)`). Hence if there is a requirement of frequent addition and deletion in application then LinkedList is a best choice.

2) Search (get method) operations are fast in Arraylist (`O(1)`) but not in LinkedList (`O(n)`) so If there are less add and remove operations and more search operations requirement, ArrayList would be your best bet.

Heap VS BinaryTree

Both binary search trees and binary heaps are both in the set of trees and are tree-based data structures. Heaps require the nodes to have a priority over their children. In a max heap, each node's children must be less than itself. This is the opposite for a min heap. Binary search trees (BST) follow a specific ordering (pre-order, in-order, post-order) among sibling nodes. The tree **must** be sorted, unlike heaps.

BST have average of O(logn) for insertion, deletion, and search.
Binary Heaps have average O(1) for findMin/findMax and O(logn) for insertion and deletion.

The insertion order for heap is fixed. Root, left child, right child.

Advantages of binary heap over a balanced BST:

- Average time insertion into a binary heap is O (log (n)). for BST is O (log (n)).
- Binary heaps can be efficiently implemented on top of arrays, BST cannot.

Advantage of BST over binary heap:

- Search for arbitrary elements O(log(n)), O(n) for heap, in which the only fast search is for the largest element O(1).

BST VS HashTable

A hash table can insert, search, delete and retrieve elements in O(1).

A binary search tree can insert, search, delete and retrieve elements in O(log(n)), which is quite a bit slower than the hash table

A hash table is an unordered data structure

A binary search tree is a sorted data structure

第一个说如果设计一个电话簿，存储所有的联系人和其电话号码，用哪个数据架构（HashTable）。

第二个说如果在给定区间内查找值，用哪个比较好（BST，因为 BST 里的是排好序的，可以利用给出的区间来更有效地查找）

So Following are some important points in favor of BSTs.

1. We can get all keys in sorted order by just doing Inorder Traversal of BST. This is not a natural operation in Hash Tables and requires extra efforts.
2. Doing order statistics, finding closest lower and greater elements, doing range queries are easy to do with BSTs. Like sorting, these operations are not a natural operation with Hash Tables.
3. BSTs are easy to implement compared to hashing, we can easily implement our own customized BST. To implement Hashing, we generally rely on libraries provided by programming languages.
4. With BSTs, all operations are guaranteed to work in O(Logn) time. But with Hashing, Θ(1) is average time and some particular operations may be costly, especially when table resizing happens.

HashTable VS HashMap

HashMap	Hashtable
1) HashMap is **non synchronized**. It is not-thread safe and can't be shared between many threads without proper synchronization code.	Hashtable is **synchronized**. It is thread-safe and can be shared with many threads.
2) HashMap **allows one null key and multiple null values**.	Hashtable **doesn't allow any null key or value**.
3) HashMap is a **new class introduced in JDK 1.2**.	Hashtable is a **legacy class**.
4) HashMap is **fast**.	Hashtable is **slow**.
5) We can make the HashMap as synchronized by calling this code Map m = Collections.synchronizedMap(hashMap);	Hashtable is internally synchronized and can't be unsynchronized.
6) HashMap is **traversed by Iterator**.	Hashtable is **traversed by Enumerator and Iterator**.
7) Iterator in HashMap is **fail-fast**.	Enumerator in Hashtable is **not fail-fast**.
8) HashMap inherits **AbstractMap** class.	Hashtable inherits **Dictionary** class.

Linked List vs Array

Both Arrays and Linked List can be used to store linear data of similar types, but they both have some advantages and disadvantages over each other.

(1) The size of the arrays is fixed: So we must know the upper limit on the number of elements in advance. Also, generally, the allocated memory is equal to the upper limit irrespective of the usage, and in practical uses, upper limit is rarely reached.

(2) Inserting a new element in an array of elements is expensive, because room has to be created for the new elements and to create room existing elements have to shifted.

Deletion is also expensive with arrays until unless some special techniques are used.

So Linked list provides following two advantages over arrays
1) Dynamic size
2) Ease of insertion/deletion O(1)

Linked lists have following drawbacks:
1) Random access is not allowed. We have to access elements sequentially starting from the first node. So we cannot do binary search with linked lists.
2) Extra memory space for a pointer is required with each element of the list.
3) Arrays have better cache locality that can make a pretty big difference in performance.

Overload VS Override

When overloading, one must change either the type or the number of parameters for a method that belongs to the same class. Overriding means that a method inherited from a parent class will be changed. But, when overriding a method everything remains exactly the same except the method definition – basically what the method does is changed slightly to fit in with the needs of the child class. But, the method name, the number and types of parameters, and the return type will all remain the same.

And, method overriding is a run-time phenomenon that is the driving force behind polymorphism. However, method overloading is a compile-time phenomenon.

Grep VS Find

Grep is used to find strings in a file. It doesn't work for directories

Find is used to find a file in a directory or to find directory in the list of directories.

Array Sorting Algorithms

Algorithm	Time Complexity			Space Complexity
	Best	Average	Worst	Worst
Quicksort	O(n log(n))	O(n log(n))	O(n^2)	O(log(n))
Mergesort	O(n log(n))	O(n log(n))	O(n log(n))	O(n)
Timsort	O(n)	O(n log(n))	O(n log(n))	O(n)
Heapsort	O(n log(n))	O(n log(n))	O(n log(n))	O(1)
Bubble Sort	O(n)	O(n^2)	O(n^2)	O(1)
Insertion Sort	O(n)	O(n^2)	O(n^2)	O(1)
Selection Sort	O(n^2)	O(n^2)	O(n^2)	O(1)
Shell Sort	O(n)	O((nlog(n))^2)	O((nlog(n))^2)	O(1)
Bucket Sort	O(n+k)	O(n+k)	O(n^2)	O(n)
Radix Sort	O(nk)	O(nk)	O(nk)	O(n+k)

Data Structure Operations

Data Structure	Time Complexity								Space Complexity
	Average				Worst				Worst
	Access	Search	Insertion	Deletion	Access	Search	Insertion	Deletion	
Array	O(1)	O(n)	O(n)	O(n)	O(1)	O(n)	O(n)	O(n)	O(n)
Stack	O(n)	O(n)	O(1)	O(1)	O(n)	O(n)	O(1)	O(1)	O(n)
Singly-Linked List	O(n)	O(n)	O(1)	O(1)	O(n)	O(n)	O(1)	O(1)	O(n)
Doubly-Linked List	O(n)	O(n)	O(1)	O(1)	O(n)	O(n)	O(1)	O(1)	O(n)
Skip List	O(log(n))	O(log(n))	O(log(n))	O(log(n))	O(n)	O(n)	O(n)	O(n)	O(n log(n))
Hash Table	-	O(1)	O(1)	O(1)	-	O(n)	O(n)	O(n)	O(n)
Binary Search Tree	O(log(n))	O(log(n))	O(log(n))	O(log(n))	O(n)	O(n)	O(n)	O(n)	O(n)

Stack:

- **Push**: **O(1)**
- **Pop**: **O(1)**

- **Top**: **O(1)**
- **Search** (Something like lookup, as a special operation): **O(n)** (I guess so)

Queue/Deque/Circular Queue:

- **Insert**: **O(1)**
- **Remove**: **O(1)**
- **Size**: **O(1)**

Binary Search Tree:

- **Insert, delete and search**: Average case: **O(log n)**, Worst Case: **O(n)**

Red-Black Tree:

- **Insert, delete and search**: Average case: **O(log n)**, Worst Case: **O(log n)**

Heap/PriorityQueue (min/max):

- **Find Min/Find Max**: **O(1)**
- **Insert**: **O(log n)**
- **Delete Min/Delete Max**: **O(log n)**
- **Extract Min/Extract Max**: **O(log n)**
- **Lookup, Delete** (if at all provided): **O(n)**, we will have to scan all the elements as they are not ordered like BST

HashMap/Hashtable/HashSet:

- **Insert/Delete**: **O(1)** amortized
- **Re-size/hash**: **O(n)**
- **Contains**: **O(1)**

Primitive Type

Type	Contains	Default	Size	Range
`boolean`	`true` or `false`	`false`	1 bit	NA
`char`	Unicode character	`\u0000`	16 bits	`\u0000` to `\uFFFF`
`byte`	Signed integer	0	8 bits	-128 to 127
`short`	Signed integer	0	16 bits	-32768 to 32767
`int`	Signed integer	0	32 bits	-2147483648 to 2147483647
`long`	Signed integer	0	64 bits	-9223372036854775808 to 9223372036854775807
`float`	IEEE 754 floating point	0.0	32 bits	±1.4E-45 to ±3.4028235E+38
`double`	IEEE 754 floating point	0.0	64 bits	±4.9E-324 to ±1.7976931348623157E+308

Logical vs. Bitwise Operators

A bitwise operator evaluates each bit of two expressions based on the logic that is defined by the operator. These include the AND, OR, NOT, and XOR operators. Each bit of the input expressions will be compared independently of other bits.

A **logical operator** is very similar to a bitwise operator in that it evaluates two conditions. IDL's logical operators are && (logical and), || (logical or), and ~ (logical not). **There are two significant differences between a logical and bitwise operator.** First, a logical operator will always return 1 (for true) or 0 (for false). Additionally, a logical operator will perform "short circuit" logic, meaning that if the outcome is known after only checking the first condition, the second condition is ignored.

Bit Manipulation:

Shift:

/* 00000001 << 3 = 00001000 */ left shift

1 << 3 == 8

0xFFFFFFF0 >> 4 == 0xFFFFFFFF right shift

The right shift operator is signed. Java, as with many languages, uses the most significant bit as a "sign" bit. A negative number's most significant bit is always '1' in Java. A signed shift-right will shift in the value of the sign. So, a binary number that begins with '1' will shift in '1's. A binary number that begins with '0' will shift in '0's. Java does bitwise operators on integers, so be aware!

You can use a third shift operator called the "unsigned shift right" operator: >>> for always shifting in a "0" regardless of the sign.

Binary "BitWise" Operations

- ~ - The unary complement 取反
- & - Bitwise and 与
- ^ - Bitwise exclusive or 异或
- | - Bitwise inclusive or 或

Java static method

Java static method program: static methods in Java can be called without creating an object of class. Have you noticed why we write static keyword when defining main it's because program execution begins from main and no object has been created yet. Consider the example below to improve your understanding of static methods.

Instance method requires an object of its class to be created before it can be called while static method doesn't require object creation.

If you wish to call static method of another class then you have to write class name while calling static method

Abstract class and Interface

abstract class

- contain at least one abstract method
- can contain numbers of concrete methods
- variable can be public , private , protected , default , or constants
- a class can only extend one abstract class
- not compulsory to implement all methods
- if want to add a new feature, simply implement in the abstract class and call it from subclass

interface

- only contain abstract methods
- variable is by default public final , only has constants
- a class can implement multiple interfaces
- compulsory to implement all methods
- if want to add a new feature, need to implement the method in all
- classes implementing the interface

final , finally , finalize()

- final : is a keyword. Final is used to apply restrictions on class, method and variable. Final class cannot be inherited, final method cannot be overridden and final variable value cannot be changed.
- finally : Finally is a block, used to place important code, it will be executed whether exception is handled or not.
- finalize() : is a method, used to perform clean up processing just before object is garbage collected.

DFS VS BFS

Both algorithms are graph traversal algorithms, and both will eventually reach every node in the graph. They differ in the order in which they visit nodes: depth first search explores all of the nodes reachable from X before moving on to any of X's siblings, whereas breadth first search explores siblings before children. In the case of a tree, depth first tree will explore one path all the way to the leaves, before backing up and trying another path, whereas breadth first will explore all nodes at depth 0, then depth 1, then depth 2, and so on, until the leaves of the tree are reached.

Memory

- BFS uses a large amount of memory because need to store pointers to a level(serious problem if the tree is very wide)
- DFS uses less memory because no need to store all child pointers at a level

Implementation

- BFS: queue
- DFS: stack

Inheritance Advantages and Disadvantages:

Inheritance allows a class to use the properties and methods of another class. In other words, the derived class inherits the states and behaviors from the base class. The derived class is also called subclass and the base class is also known as super-class. The derived class can add its own additional variables and methods. These additional variable and methods differentiates the derived class from the base class.

Advantages:

1. Minimize the amount of duplicate code in an application by sharing common code amongst several subclasses. This also tends to result in a better organization of code and smaller, simpler compilation units.

2. Inheritance can also make application code more flexible to change because classes that inherit from a common superclass can be used interchangeably. If the return type of a method is superclass

3. Overriding--With inheritance, we will be able to override the methods of the base class so that meaningful implementation of the base class method can be designed in the derived class.

Disadvantages:

1. One of the main disadvantages of inheritance in Java (the same in other object-oriented languages) is the increased time/effort it takes the program to jump through all the levels of overloaded classes. If a given class has ten levels of abstraction above it, then it will essentially take ten jumps to run through a function defined in each of those classes

2.Main disadvantage of using inheritance is that the two classes (base and inherited class) get tightly coupled.
This means one cannot be used independent of each other.

3. Also with time, during maintenance adding new features both base as well as derived classes are required to be changed. If a method signature is changed then we will be affected in both cases (inheritance & composition)

Inheritance vs Composition

By composition, we can simply use instance variables that are references to other objects

1) Static vs Dynamic

First difference between Inheritance and Composition comes from flexibility point of view. When you use Inheritance, you have to define which class you are extending in code, it cannot be changed at runtime, but with Composition you just define a Type which you want to use, which can hold its different implementation. In this sense, Composition is much more flexible than Inheritance.

2) Limited code reuse with Inheritance

As I told, with Inheritance you can only extend one class, which means you code can only reuse just one class, not more than one. If you want to leverage functionalities from multiple class, you must use Composition.

3) Unit Testing

When you design classes using Composition they are easier to test because you can supply mock implementation of the classes you are using but when you design your class using Inheritance, you must need parent class in order to test child class. There is no way you can provide mock implementation of parent class.

5) Encapsulation

Last difference between Composition and Inheritance in Java in this list comes from Encapsulation and robustness point of view. Though both Inheritance and Composition allows code reuse, Inheritance breaks encapsulation because in case of Inheritance, sub class is dependent upon super class behavior. If parent classes changes its behavior than child class is also get affected. If classes are not properly documented and child class has not used the super class in a way it should be used, any change in super class can break functionality in sub class.

Polymorphism and encapsulation

Polymorphism is the ability to create a variable, a function, or an object that has more than one form.

In a given class you can have methods. When you have two methods with the same name, but different parameters in the same class, you have polymorphism. Sometimes this kind of polymorphism is called overloading.

Another way of having polymorphism is when you have a method in a class and you reimplement it differently in a subclass. So, for the very same name and parameter list you have two different behaviors in two different (but related) classes.

Advantages of polymorphism:

- -Same interface could be used for creating methods with different implementations
- -Reduces the volume of work in terms of distinguishing and handling various objects
- -Supports building extensible systems
- -Complete implementation can be replaced by using same method signatures

Encapsulation (data hiding) is a technique used for hiding the properties and behavior of an object and allowing outside access only as appropriate. This prevents other objects from directly altering or accessing the properties or methods of the encapsulated object.

Object VS Class

No.	Object	Class
1)	Object is an **instance** of a class.	Class is a **blueprint or template** from which objects are created.
2)	Object is a **real world entity** such as pen, laptop, mobile, bed, keyboard, mouse, chair etc.	Class is a **group of similar objects**.
3)	Object is a **physical** entity.	Class is a **logical** entity.
4)	Object is created through **new keyword** mainly e.g. Student s1=new Student();	Class is declared using **class keyword** e.g. class Student{}
5)	Object is created **many times** as per requirement.	Class is declared **once**.
6)	Object **allocates memory when it is created**.	Class **doesn't allocated memory when it is created**.
7)	There are **many ways to create object** in java such as new keyword, newInstance() method, clone() method, factory method and deserialization.	There is only **one way to define class** in java using class keyword.

Process VS Thread

Process	Thread
A process has separate virtual address space. Two processes running on the same system at the same time do not overlap each other.	Threads are entities within a process. All threads of a process share its virtual address space and system resources but they have their own stack created.
Every process has its own data segment	All threads created by the process share the same data segment.
Processes use inter process communication techniques to interact with other processes.	Threads do not need inter process communication techniques because they are not altogether separate address spaces. They share the same address space; therefore, they can directly communicate with other threads of the process.
Process has no synchronization overhead in the same way a thread has.	Threads of a process share the same address space; therefore synchronizing the access to the shared data within the process's address space becomes very important.
Child process creation within from a parent process requires duplication of the resources of parent process	Threads can be created quite easily and no duplication of resources is required.

TreeMap

TreeMap is an example of a SortedMap, which means that the order of the keys can be sorted, and when iterating over the keys, you can expect that they will be in order.

All operation is O(log n)

synchronized VS volatile

synchronized:

- provide lock on the object and prevent race condition
- can be applied to static/non-static methods or block of code
- only one thread at a time can access synchronized methods
- if multiple threads, need to wait for execution complete

volatile:

- volatile variable stored in main memory
- every thread can access
- local copy updated from main memory
- has performance issues