Interview Preparations Documents

+ Search

- Games usually are PSPACE complete
  NP-complete → PSPACE
- Complete Search Algorithm: Guaranteed to find a solution or prove even if there is none. It will prove something
- Incomplete: May not find a solution even if one exists. often more efficient.

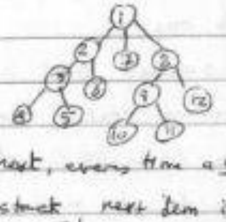
- A search state is a partial step in the search process that may specify everything about a possible solution or may not solve the problem or lead to one.
- Search tree is a representation of the search space, the nodes are the search states, links (edges) are logical (abstraction) connections between search states
→ Abstraction of one possible search
→ do not summarise all possible searches
→ Root is null state, childs are extensions, leaf nodes are solutions or failures.
→ Algorithms do not store whole search trees, needs exponential space nodes already explored can be discarded.
→ Search Algorithms stores the search frontier (nodes with some unexplored children)

① Depth-first search: Pick deepest left most element of frontier
explore all nodes in subtree of current node → move.
② Breadth-first Search: Pick shallowest left most node of frontier
explore all nodes at one height
③ Best first search: whichever element seems promising
④ Depth-bounded Depth-first: like depth but with limit on depth
⑤ Iterative Deepening: Depth-bounded but increase limit iteratively.

- lists can easily store the search frontier.
- Each element in the list is a search state
- Different Algorithms manipulate lists differently.

* Pseudo code for Depth first Search (DFS)

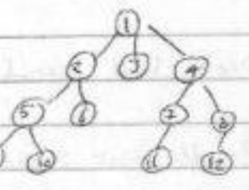
```
Set S = {(s)}    set of explored vertices
stack T = All neighbours of (s)
While [ T is not empty ]
        pop vertex v from end of T    (stack)
        if (v is not in s)
            add v to s
            push neighbors of v onto the end of T
        end if
end while
end.
```



DFS uses a stack to determine which vertex to visit next, every time a new vertex is visited all its neighbours are added to the top of stack, next item is the pop (first element) of the stack [LIFO] $[O(n^2)]$

* Breadth First Search (BFS)

```
Set S = {s}    set of visited vertices
Queue q = all neighbours of s
while (q is not Empty)
    de-queue vertex v from the front of Q
    if (v is not in s)
        add v to s
        add to queue neighbours of v to the q
    end if
end while
```



BFS uses Queue as its data structure to determine which vertex to visit next every time a node is visited, neighbours are added to the ~~top~~ end of the queue next item is the front of the queue (last one entered) $\left[\begin{matrix} O(2^n) \\ \text{exponential growth} \end{matrix}\right]$

* Depth first depth bounded; same as DFS but with limited depth.

DFS may find very deep solutions before shallow ones.

DFDB is relevant if tree contains infinite branches

won't get stuck in cycles, guarantee to find solution if within depth (complete)

[Recursion]

* **Factorial:**

```
public int factorial (int n) {
        if (n == 0) return 1;
        else { return n * factorial (n-1); }
```

* **Fibonacci:**

```
public int Fib (int n) {
        if (n==0) || (n ==1) return 1;
        else return fib(n-1) + fib(n-2); }
```

[6 years ago
sketchup
3D buildings site
google earth
& comp science + Math

[0 1 2 3 +]

Pseudo codes

* **Factorial:**

```
function factorial as:
    input : integer n such that n >= 0
    if n is 0 , return 1
    otherwise return [n x factorial (n-1)]
    end factorial
```

* **Fibonacci:**

```
Function Fib is :
    input : integer n such that n >= 0
    if n is 0 return 0
    if n is 1 return 1
    otherwise return [ Fib (n-1) + fib(n-2)]
    end Fib
```

* **Greatest common divisor:**

```
Function gcd is:
Integer X, integer Y such that X >= Y and Y>= 0
if y is 0 return X
otherwise return [ gcd (y, remainder of X/Y)) ]
```

* It is important to define dynamic data structures such as Lists and Trees, they can dynamically grow in response to runtime requirements.

* Towers of Hanoi

32 bits per pixel → 8 bits

pallate

256 entries

each entry 4 bytes

input

4 bytes per pixel

1 red byte

1 Green byte

1 Blue Byte

1 K Byte

4 Bytes

pixel → 4 bytes

4

pixel

width & hight

width * hight * 4 =

###

4[

	Best	Worst	Average
Exchange Sorts:			
Bubble Sort	$O(n)$	$O(n^2)$	$O(n^2)$
Quick Sort	$O(n \log n)$	$O(\cancel{n \log n})$ $O(n^2)$	$O(n \log n)$
Selection Sorts:			
Selection Sort	$O(n^2)$	$O(n^2)$	$O(n^2)$
Heap Sort	$O(n \log n)$	$O(n \log n)$	$O(n \log n)$
Insertion Sorts:			
Insertion Sort	$O(n)$	$O(n^2)$	$O(n^2)$
Tree Sort	$O(n)$	$O(n \log n)$	$O(n \log n)$
Binary tree	$O(n)$	$O(n \log n)$	$O(n \log n)$
Merge Sorts:			
Merge Sort:	$O(n \log n)$	$O(n \log n)$	$O(n \log n)$
Distribution Sorts:			
Bucket Sort			

- Best first + Garanties = A^*
  Depth first + Guranties = BnB

- Hill Climbing: Pick a node S, choose from neighbours node S' where S' is the highest neighbour, iterate until S' is the highest point.
- → Hill climbing first choice: 1st higher neighbour
- → Stochastic: choose random neighbour
- → Sideways moves: when the __best__ successor has the __same__ value as the current state, this allows getting off the plateau without looping.
- → Simulated Annealing: Stochastic hill climbing with downward moves allow

- __Local Beam search:__

Keeping one node in memory is extreme to deal with memory limitations. Local beam search keeps track of K states rather than just one, it begins with K randomly selected states, at each step all K's successors are generated if any is the goal, done else select the K best successors and repeat.

- What's the difference with Random Restart?

RR each process run independently of the others, local beam useful information is passed among the K parallel threads.

__ex:__ thread K generates good successors, other K-1 threads generates bad (K-1) abandon their unfruitful searches and move to where the best progress is made.

→ LB can suffer lack of diversity among K states
   Become easily concentrated in a small region of state space making search more expensive than hill climbing

↘ Stochastic LB: pick random K

Pseudo code: Depth first B&B

① create a list P, best is ∞
② Add the start node S to P so P = [S]
③ while P is not empty
 ④ Extract the first element S from P
 ⑤ if S = goal , set Best = cost (goal)
 ⑥ Extend S to all neighbours
 ⑦ Examine all paths
 ⑧ Reject paths with loops or cost ≥ best
 ⑨ add remaining to the start of list P using depth
end if
end while
If goal found → return best else nothing found

S → A = 0 + 3 = 3 ✓, S → D = 0 + 4 = 4 S → A
A → B = 3 + 4 = 7, 7 + 0 = 7 ✓, A → D = 8 + 0 = 8 S → A → B

S = 0 + 0 = 0
S → A = 0 + 3 = 3 S → D = 0 + 4 = 4
S → A → B → 3 + 4 = 7 S → A → D = 3 + 5 = 8
S D = 4
S → A → B → E = 7 + 5 = 12 ~~S → A → B → C~~
S → A = 0 + 3 = 3, S → D = 0 + 4 = 4
S → A → B = 7 + 0 = 7, ~~S → A → D = 8 + 0 = 8~~, SD → 4
① [SAB = 7, SD = 4
SABC = ~~11 + 0 = 11~~ loop, SABE = 12 + 0 = 12, SD = 4
② [SABE = 12, SD = 4 ...

→ this example is long because the heuristic is bad and the estimate (distance to goal is always at least 0) is bad.
→ Improve by adding a non-zero estimate
 total underestimate = current distance travelled + underestimate of distance remaining
* Underestimate must hold the shortest possible path

$$\sum_{i=1}^{n} i = n(n+1)/2$$

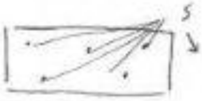
Optimize time

* Robot to assemble a circuit:

  → Robot arm move at constant speed

  → the time taken depends relatively on the distance

↘



S points

↘ shortest cycle to visit each point at the board in the set S

* Nearest Neighbor:

  ① from a starting point visit the nearest neighbor

  ② from the neighbor visit the nearest unvisited neighbor

```
Pick and visit initial point P0 from P
P = P0
i = 0
while there are still unvisited points
        i = i + 1
        select Pi to be the closest unvisited point to Pi-1
        visit Pi
return to P0 from Pn-1
```

It looks at each pair at most twice (i, j) when adding P_i and when adding P_j

[Design Patterns]

* Singleton:

- only one Istance of class is allowed, when creating an object and one already exists → return back the object reference.

```
public class Singleton
        Private static final Singleton INSTANCE = new Singleton();
    private Singleton(); { }
    public static Singleton getInstance() {
            return Instance; }
```

Dis-Adv

→ + Makes unit testing more difficult
- Introduces global state for application
- Reduces parallelism potential

* Factory: creating objects without specifying the exact class of object that will be created.

→ abstraction of a constructor

* Google search enhancement

→ site search
Image search (right click → add to picasa)
in URL
Dictionary

* Immutable class in JAVA

→ all its field final
class declared as final
Any data which refer to mutable object is:
- private
- no setter method

* TreeSet Elements must be comparable

Tree set overloaded constructor that takes comparator.

* Operators

int x = 10 ;   ~X   bitwise NOT operator
flips all bits
0000 1010
~X 1111 0101 → 10 → -10   Negation

& | compare bits

X = 10 // 00001010
X = 6 // 00000110

X & Y → 0000010
X | Y → 0000111 0

[XOR exclusive or]   X ^ Y → 0000 1100   return true only if one element is true

X >> 2
→ shift to the right   ✗2✗2 = x4

X << 2 left shift
X 2 ...

Binary numbers have the left most digit as sign bit

1 → negative   0 → positive

>>> unsigned right shift → fills the left with zeros

Public → anywhere   Inherit from it
protected → like default but outside package can ~~[illegible]~~
default → only within the package
Private → within the same Class

enum member { A, B, C };

* Scheduling Algorithms :

$\boxed{\text{Map Reduce}}$

- More processes than machines
→ centralized machine that assign tasks
→ Assign tasks to machines that are near the input data
(locate computation on or near input data)
→ preserve Network bandwidth
→ Recovery from failure

* Caching
→ If a task takes so much time, Master will make the same task run on another machine in case of losts.

* Communication coordination (Bandwidth, Processing speeds)

$\underbrace{\text{large data / lots of computation}}$
libraries spread it across machines, deal with failures → aggregate results
ex: counts pages on different language
function to identify a page's language
→ Map reduce → function on large set → results.

* Google File System → for large distributed data intensive apps.
→ fault tolerance while running on inexpensive hardware
→ delivers high aggregate performance to large clients.

* Google Fusion Tables
Google Goggles
* Google Correlate ✓ find search patterns corresponds to real world trends
Google news time line

Google Plus+

Good:
Circles / Activity streams
Feedback Mechanism
Hangouts
Sports → Add Interest / passions (supersede Google Alerts)
Huddle

Bad:
Brand Pages / Business Pages
Google Reader / Buzz Integration
If I post sth to a circle, someone can re-share it outside
→ sharing topics, keeping track on comments
→ Subscribe to one guy but filter some of his content, # tags

Google Scribe

* Integration into google Docs
* Publish at Blogger, attach from gmail
* Suggest synonyms
* Export to PDF
* Learn new words / add to Dictionary
    ↳ It learns in the document but not after exit

Youtube

* No. of subscribers who watched a movie I published
* Filter and clean videos (no views ... etc)
* Copyright help
* upload more videos
* Schedule upload, or upload and put live later
* TV / youtube guide
* Recommend channels
* Home Page
* Video categories
* Subtitles
* Hang when Full screen
* channel customization (Background ... etc)

Google Cal

* Sync with G+ contact or FB for birthdays
* FB events
* Schedule Hangouts
* Scheduling → Pending proposed times when busy
    schedule on behalf of others
    schedule resources
    schedule in any calendar
    Add dates to a new cal → add FB Birthdays to main?
    Holidays, Soccer ... etc
    one - on - one scheduling
    Activity scheduling
    Group scheduling
- Alerts

Google Docs

* Allow use of all google fonts across All Docs tools
* Better function for header / footer, page numbering
* Available in gmail as attachments
→ available apps
* themes for presentation
* ~~add diagrams & graphs to Docs.~~
* More formatting Tools
* Basic edits
→ Mobile support
* Save As
→ Import files more than 1MB

Google scholar

Add more fields to refine search → Author, publisher .... etc

Google Finance

* Show historical graph of portfolio
* Alerts based on Stock, option ...etc
* Show % of outstanding stocks held by investors
* Historical Data
* Ability to click on different Financial terms and get background on them
* View the price / value history of options, just like stocks
* Multiple currency converter
* Adjustable portfolio
* Analyze the value of Business, discover the Intrinsic value of Stocks
  → Analyze Financial Statement
  → Reporting
  → Portfolio valuation

Picase:

* Contacts Manager Integration with picasa and customizable Facebook-like tags
  → tags directly / Maybe Auto Detect
  [Google has the edge over Facebook as you can geo-tag photos and display them in google Maps]

* Picase quota that increases as your items are viewed more
  → view, download, comment → Increase space
  Default 1 GB

* Integration of google Docs, picasa with gmail
  ↳ attach directly from gmail.

→ Picase Art room → IBM Gallery of me

* Sharing Ads

Google Images

* Second link to skip directly to full version
* Upload an image → find all other version and sizes $\xrightarrow{\text{like}}$ TinEye
* Draw a sketch and find real Images that looks like it
* Adapt Display results like Web results
* Search by location using geo tags
* Trending Images
* Facial Recognition
* Personally remove certain pictures and not see them again
* Adult Filter
* Support copyrights by photographer

ASCII to String: String s = new Character ((char) 64). toString();

* Enhanced for Loop: for (int n $\overset{in}{:}$ Array)

loop through the array and assign each time a value to n until there are no more elements.

* Random Number Generator: int r = (int)(Math.random() * 5)

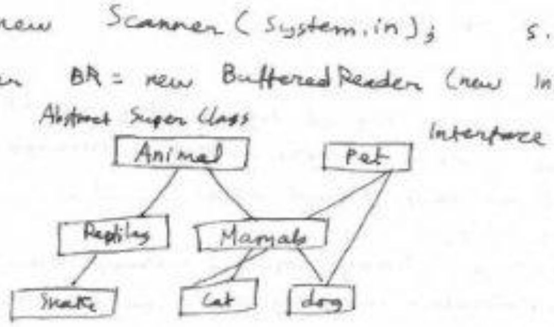
- random returns a double → cast to Integer
- the value returned by the random function is between 0, 0.999...
  to extend the range multiply by another number.

* Read Input

① Scanner s = new Scanner (System.in);   s.readline(); ....

② BufferedReader BR = new BufferedReader (new InputStreamReader (System.in

* Polymorphism



- Polymorphism is achieved by Animal a = new Dog();
- The class animal is a class that defines behaviours of other classes, but I will not go and have an instance of class Animal →

* Abstract Class: Must be extended
  Abstract Method: Must be overriden
  Abstract Methods $\xrightarrow{\text{exists in}}$ Abstract classes
* Final class: I can't extend it, no subclasses from it, no Inheritance
* Multiple Inheritance is not allowed in Java → Interfaces
  → I can implement multiple Interfaces
  Interface: treat object by its behavior not Inheritance.
* Static Classes, methods can be accessed without instantiating new objects
  → If I am going to call a method → static

* **Java Data types :-**

Integer
- Char : 16 bits — 0 → 65535
- Byte : 8 bits — -128 → 127
- Short : 16 bits
- Int : 32 bits
- long : 64 bits

Floating
- float : 32 bits
- double : 64 bits

Java Defaults:
- byte, short, int : 0
- long: 0L
- float : 0.0F
- double : 0.0d
- char: '\u0000' (space)
- String : null
- boolean: false

$2^0 \to 1$
$2^1 \to 2$
$2^2 \to 4$
$2^3 \to 8$
$2^4 \to 16$
$2^5 \to 32$
$2^6 \to 64$
$2^7 \to 128$
$2^8 \to 256$
$2^9 \to 512$
$2^{10} \to 1024$

$2^{11} \to 2048$
$2^{12} \to 5096$
$2^{13} \to 8192$
$2^{14} \to 1158$
$2^{16} \to 65536$

* In Defining float in Java I have to use (F) as Java interprets anything with a decimal point as double

float F = 32.5F ;

* Arrays in Java cannot be resized dynamically in runtime

solution :- ① use lists or Vectors instead of Arrays

list e = ~~Kaka~~ Arrays. asList ( ) ;

② use Copy of

Int[] a = Arrays. copyOf (oldArray , New Size);

③ Re- allocate a new array with a new size and copy contents

→ to do it manually Keep in mind :
- * the old and new size at the array
- * the data type of the array elements

* When converting between data types → from sth1 to sth2 → sth2. ⎵

Double to string	String s = Double. toString (d);
Long to String	Long . toString (F) ;
String to Integer	int i = Integer. valueOf (string) . intvalue();
	Integer. parseInt (str);
	Long. parseLong (str) ;
	Double. parseDouble (str, default);
	if str, result is invalid
Decimal to binary:	String s = Integer . toBinaryString(i) ;
to hexadecimal	Integer . toString (i , 16) ; Base
	Integer . toHexString (i);

Sorting Algorithms:

typical good behaviour is $O(n \log n)$
bad behaviour is $O(n^2)$
ideal behaviour is $O(n)$

comparison based sorting Algorithms need at least $O(n \log n)$

$\boxed{\text{* Bubble Sort}}$ Worst case: $O(n^2)$
Best case: $O(n)$
Average case: $O(n^2)$

→ go through list → compare pairs and swap

Advantage [not over Insertion Sort]
can detect if the list is sorted
→ some algorithms like quick Sort perform the process on the whole set even if it is already sorted.

How to optimize it?

Generally after every pass all elements after the last swap are sorted and do not need to be checked again

$\boxed{\text{* Selection Sort}}$ Worst, Best, Average case $= O(n^2)$
→ has some advantages esp. when memory is limited

Algorithm:
1. Find the minimum value in the list
2. Swap with the first
3. Repeat

$\boxed{\text{• Insertion Sort}}$ Worst - Average $O(n^2)$, Best $O(n)$

→ The sorted array or list is built one entry at a time

Advantages:
1. Simple Implementation
2. Efficient for small data sets
3. Adaptive [for data that is semi sorted or sorted]
4. Stable: doesn't change the order of elements with equal keys
5. In-place: requires constant amount $O(1)$ of memory space
6. On-line: can sort a list as it receives it.

→ Not Efficient on large sets
→ take an element and its adjacent, swap if smaller and go next
→ check next with the whole previously sorted elements!

+ Merge Sort | Best - worst - Average → $O(n \log n)$

① If the list is of length 0 or 1 then it is already sorted
else ② Divide the unsorted list into two sublists // half size
③ Sort each sublist recursively // re-apply merge sort
④ Merge the two sublists back into one.

→ small list takes fewer steps to sort
→ fewer steps required to construct a sorted list from two sorted sub lists.

Optimization | cache aware version
→ stop partioning when reaching specific subarray size S
S is the CPU cache size
+ Parrallize the recursive division of the array + merge

+ HeapSort | Worst, Best, Average $O(n \log n)$

+ Building a heap, take the largest element, re-construct ... etc
Max-heap or Min-heap
+ Requires two arrays, one to hold the heap, one to hold the sorted elements
+ Quick sort is faster but with worst case of $O(n^2)$

+ Quick Sort

+ Pick an element, called Pivot from the list
+ Re-order elements in list so that : before pivot → smaller, After Pivot → greater] Partitioning
+ Recursively sort the lower, greater elements.

+ Bucket Sort | Average $O(n+K)$, Worst $O(n^2)$

+ Partition the array into buckets, each is sorted individually.
→ ① Set-up an array of initially empty buckets
② Scatter: Go over the original array, putting each object in its bucket
③ Sort each non-empty bucket
④ Gather.

* LinkedHashSet

→ Guarantees the Iterator will return their elements in the order which they were first added. How?

maintaining a linked list of the set elements.

→ Faster to traverse but overhead

→ choose this if the order or the efficiency of Iteration is important

* CopyOnWriteArraySet

→ based on an array that is treated as immutable

a change to the contents of the set result in an entirely new array being created.

→ Not suitable if am expecting many searches or insertions

But Iteration cost $O(1)$ → faster than HashSet

Adv: provides Thread Safety without adding to the cost of read

why? read operations are implemented on the backing array which is never modified after its creation.

no interaction from write thread.

* SortedSet

→ Iterator will traverse the tree in ascending order

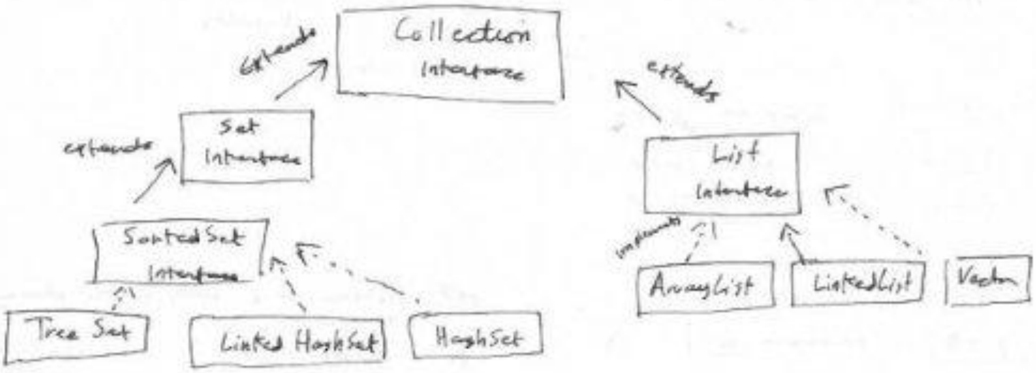
Merging two sorted lists of size n is $O(n)$

adding n elements to a tree set of size n is $O(n \log n)$

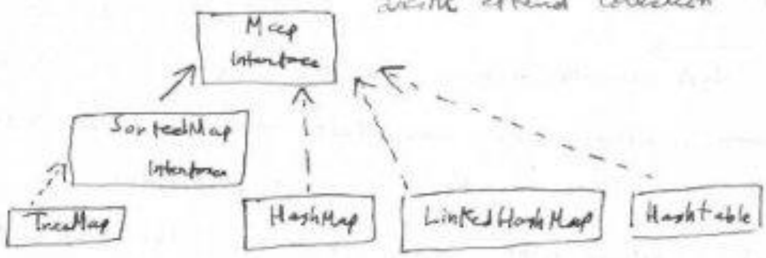
* Navigable Set [preferred over Sorted] i.e PollFirst, PollLast
Introduced new methods Range Views

* Concurrent SkipList Set → based on linked lists

* Use $\boxed{\text{list}}$ when $\boxed{\text{Sequence}}$ Matters → index position
more than one element referencing the same obj → duplicates
* Use $\boxed{\text{Set}}$ when uniqueness matters → No duplicates are allowed
* Use $\boxed{\text{Map}}$ when finding sth by Key matters
IMP No Duplicate $\boxed{\text{Keys}}$, ok to have duplicate values



doesn't extend Collection



* when Using different sets for custom classes I have to override the hashcode and equal methods to make sure the comparison for the sort works in a good way.

two Objects are equal → Same Hashcode

Same Hash Code → not necessarily equal

```
Public boolean equals (Object song) {
                                    }
public int hashcode () {
            return title.hashcode();
```
→ string has already overridden hashcode method

[Maps]

- Key, Value pairs
- HashMap provides constant time performance for put and get
→ this is guaranteed (close to guarantee) when there are no collisions but it can be closely approached by re-hashing to control the load

- Linked Hash Map → Guarantees the order [Inserted or accessed]
- WeakHashMap →
  - Normal HashMaps keeps "strong" references to all objects
    → even when a Key becomes unreachable it cannot be garbage collected.

  But
  If the objects of the Key class are unique [object equality is the same as object identity], each Object might contain a unique serial no. → so, once we no longer have reference we can never look it up again because we cannot re-create it.

- IDentity HashMap
  - two Keys are considered equal only if they are physically the same object.
  → used in Serialization
- SortedMap
→ traversal in Ascending Key order
- ConcurrentMap
→ In high performance server applications as cache implementations

- Google MultiMap → multiple values for the same Key

* Red Black Trees   Search, Insert, delete $\rightarrow O(\log n)$

+ Self balancing binary search tree, re-balanced in $O(\log n)$
+ leaf nodes do not contain data, to save memory a sentinel node performs the role of all leaf nodes.
* Allow efficient in-order traversal

Properties

① A node is either red or black
② The root is Black [In some implementation not necessarily]
③ All leaves are black
④ Both children of every red node is black
⑤ Every simple path from a given node to any of its descendants leaves contains the same no. of black nodes

this implies
$\longrightarrow$ path from root to the furthest leaf is no more than twice as long as the path from the root to the nearest leaf $\longrightarrow$ roughly balanced

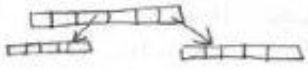
[they offer worst case guarantee]

Applications:

+ Valuable in time-sensitive Apps, real-time applications
Maybe schedulers

+ N-Array trees:

If we relax that each node can have only one key, we can reduce the height of the tree.



① All leaves are on the same level
② All nodes except the root and leaves have at least $m/2$ and at most m children. Root at least 2, at most m. tree order

* AVL Trees:

$O(\log n)$ Insert, remove, search

More Rigidly balanced than red-black → slower insertion and removal
Faster retrieval

→ Good for data structures that are built once without re-construction
→ language dictionaries

* The heights of the two child subtrees differ by at most one.
* Balance Factor = Height of left subtree − height of right subtree
Any node with balance factors of 0, 1, −1 → balanced
* Insertion is done by checking the balance factor.

* SPLay tree

	Avg	Worst *
Insert, remove, search	$O(\log n)$	$O(n)$

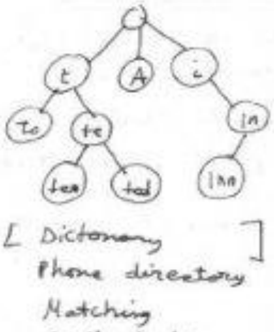
* All operations are combined with one an operation called splaying → re-arranging the tree so that the element is placed at the root.
→ Accomplished By: Search + tree rotations
Having frequently-used nodes near the root is an adv.
specially to implement cache + garbage collectors

Disadvantage: The height of the tree can be linear
↘ accessing all elements in non-decreasing order

* Trie-tree

Key_value

* Used to store associative Array where keys are usually strings
→ the position in the tree shows what key it is associated with.
+ Descendents of a node have common prefix of the string of that node and the root is associated with empty string.



→ time to insert, delete, find is almost identical
→ better CPU and branch caches, better than hash tables and binary search trees.

→ looking up a key of length m takes $O(m)$ / Binary Search Tree makes $O(\log n)$ comparisons, depends also depth $= O(m \log(n))$
→ more space efficient
→ support ordered iteration

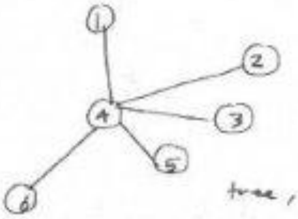
[Dictionary]
Phone directory
Matching
Spell checking

[Trees]

- Fast insertion, retrieval for the data in <u>ORDER</u>

  <u>ex:</u> Matching a word against a prefix
  I don't use hashtable as I can't retrieve data by content and they are not fast enough

* Height: length of the longest downward path from leaf to that node
  → the height of the root = height for the tree
* Depth: length of the path to the root [0 depth → one node ; -1 depth → no nodes]
* No cycles are allowed in a tree

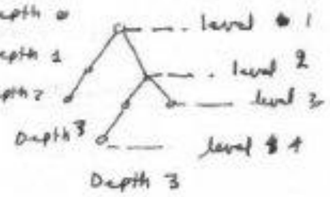


Vertices $V = 5$
Edges $V-1 = 4$

tree, connected, no cycles

* <u>Binary trees</u>

  - Has at most two childs for a node
  - Tree Depth = Tree level − 1



  * Rooted binary tree: tree with root, every node has at most 2 children

  - Full Binary tree (2-tree): every node other than leaves has two children
  - Perfect Binary tree: Full Binary tree with all leaves at the same level
  - Complete Binary tree: Every level maybe except the last is completely filled and all nodes are as left as possible

  - Balanced Binary tree: no <u>leaf</u> is much further away from root than any other leaf.
    The depth for BBT = $\log_2(n)$ ← number of nodes

Number of Nodes in: ① Perfect Binary tree → $2^{h+1} - 1$ (h: height)
② Complete Binary tree → min: 2^h max $2^{h+1}-1$

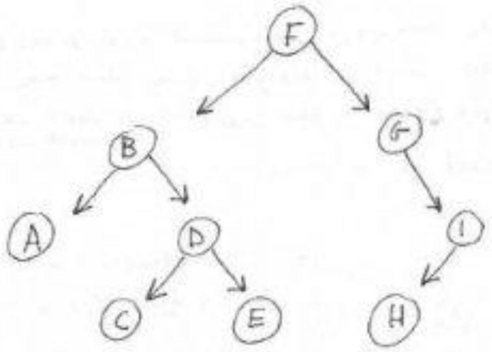
" " leaf nodes in complete BT $n/2$
" " " " Perfect BT 2^h

N

* Tree traversal :

1. Pre- order : Root, left, right
2. Post- order : left, right, root
3. In- order : left, root, right

→ Depth First, Breadth First



Depth

Pre-order : F, B, A, D, C, E, G, I, H

In- order : A, B, C, D, E, F, G, H, I

Post- order: A, C, E, D, B, H, I, G, F

Breadth (level traversing)

F, B, G, A, D, I, C, E, H

* In an ordered binary tree, left node keys are less than root
right node keys are greater than root

* Bounded Queue → Queue limited to a fixed No. of items.

* Concurrent Linked Queue

  + Non-blocking Queue, unbounded and thread safe.
  + Uses a linked structure → Insertion and removal in constant time
  + They are the basis for skip lists

* Blocking Queue

  * Designed primarily for consumer - producer queues.
  + Ex: Print spooling : Add print Jobs, they are processed one by one
    a print server doesn't need to constantly poll the queue to know whether jobs are waiting
  * A timeout can be defined

* Linked Blocking Queue

  + Thread safe based on linked node structure

* Array Blocking Queue

  + Implementation Based on a circular Array [first and last elements are logically adjacent]
  + Each time the head is removed, head index is incremented

+ Priority Blocking Queue

  + Thread safe, blocking version of Priority Queue.

+ Delay Queue

  + Ordering is based on the delay time for each element
    the time remaining before the element will be ready to be taken from the queue.
  + Positive + → timer not expired [Peek will allow to see the first unexpired item]
    negative - → timer expired → poll(): least negative [longer delay]

+ Dequeue

  + Double ended queue, insertion and removal at both ends
    taking element from head → FIFO, from tail → LIFO

[Queues]

* FIFO order → First IN First out
* Data is stored to be processed up later
* Queue performs the function of Buffer
* Normal Queue Attribute : Remove elements from front
  Add elements to the back
* Usually no size is defined → dynamic adding / Removing as re-sizing
* When Implementing Queue as an Array I will need a variable that will store the value of the First element index and one that will store the size
  → why don't we store the value at the front of the Queue, always at index ∅
  Because then, every time we de-queue we have to move every element to the previous index, this takes $O(n)$ time.
  → If I am storing the index of the first element what happens if I run out of room and I want to wrap over?
  → that's why I keep a variable that Stores the no. of elements.
  → I can find the last time index → first item index + no. of elements

---

* Priority Queues:
  * Each element is Associated with a priority
  * Elements are pulled highest priority first
  → Implementations:
  ① Naiive: keep elements unsorted, upon request search through all elements
  Insertion → $O(1)$ Retrieval → $O(n)$
  ② Sorted list: Important elements first
  Insertion → $O(n)$ Retrieval $O(1)$ Initialization using Quick Sort $O(n \log n)$
  ③ Heaps: This gives $O(\log n)$ insertion & removal
  ④ Self-Balancing Binary Search Tree: $O(\log n)$

Usage: ① Bandwidth Management
② Dijkstra's Algorithm [extract minimum when graph is stored in adjacency list or matrix]
③ A^* Algorithm: used to keep track of unexplored routes

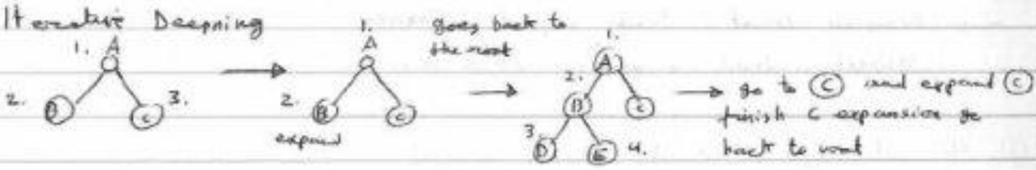
* Not designed for concurrent using → not thread Safe, no blocking behaviour

- Iterative deepening finds the best limit by gradually increasing it until the goal is found.
  → combines benefits of Depth and breadth.
  like depth in memory requirement $O(bd)$
  like breadth: complete when the branching factor is finite and optimal when the path is non-decreasing function of the depth of the node.
- May seem wasteful because search states are generated multiple times, but this is not too costly. why?
  In a search tree with the same (nearly the same) branching factor at each level, most of the nodes are in the bottom level, so it doesn't matter much that the upper levels are generated multiple times

- Memory requirements are a bigger problem for breadth first than is the execution time.
- Exponential complexity search problems cannot be solved by uninformed methods for any but the smallest instances.

- Iterative deepening is the preferred uninformed search method when the search space is large and the depth of the solution is not known

Iterative Deepening



root, down one level, expand it all, root, one more level down, expand it all.

- Informed search can find solution more efficiently.

- Greedy Best first tries to expand the node closest to the goal.
→ search cost is minimal but it is not optimal.
→ each step it tries to get to the goal as close as it can.
→ incomplete even in a finite state space
time and space $O(b^{m})$ → maximum depth

- Searching using A^* is optimal if $h(n)$ is admissable

* Local search:
suitable for problems in which all what matters is the solution state not the path cost to reach it.
→ we relax assumptions
- Online search: state space initially unknown and must be explored
8 queens problem: what matters is the position of the queens not how they were placed.
- They operate using a single current node (not multiple paths) and move to neighbors of that node. Paths are not retained.
① Very little memory (often constant)
② find reasonable solutions in large or infinite state spaces.
→ good to solve optimisation problems
→ complete local: finds a goal if exists
optimal: find a global max or min

① Hill climbing: (Greedy local Search)
loop that continually moves in the direction of increasing value, terminates when reach a peak, no neighbour has higher value.
→ No search tree, No data structure
→ doesn't look beyond immediate neighbours
→ if there is a set of best successors, randomly pick one
→ grabs a good neighbor without thinking ahead.
→ local maxima, peak higher than all neighbors but lower than global maxima
→ Ridges: sequence of local maximas (hard to navigate)
→ plateaux: no uphill exit exists, no progress
stuck when best successor has same value as current state

* Iterating over a hashtable requires each bucket to be examined to see if its occupied or not

→ time cost proportional to the capacity of the hash table ⊕ the number of elements it contains

* Data Structures *

- Arraylist
- TreeSet : Keeps the elements sorted and prevents Duplicates Insertion can be expensive
- Hashmap: store values as (Key, Value) pairs
- Linkedlist : better performance on Insertion, deletion ... etc
- Hashset : Prevents duplicates
- Linked HashMap : HashMap + remmeber the order in which elements were inserted or order of which elements were last accessed.

→ Sort an array list : Collections. Sort (Arraylist) ;

From array to ArrayList

Another Collection → Array to list

ArrayList<Int> arr = new Arraylist <Integer>(Arrays. asList (array));
~~ArrayList~~

Public <T extends Animal> void A (ArrayList <T> Arraylist)

* To be able to compare stuff in JAVA

① The class to implement comparable

→ Invoke the one argument sort (List l)

→ the compareTo overriden method decides how

ex: class Song implements Comparable < Song > {

public int CompareTo (Song s) {

return title . compareTo (s. getTitle()); }

→ I can only compare to one criteria

② I do not implement Comparable, I create new class that implements comparator and I pass it to the sort (List l, comparator c)

ex: class ArtistCompare implement comprator <Song> { → we are inside Song class

public int compare (Song one, Song two) {

return one. getArtist(). compareTo (two. get Artist ()); } }

[Sets]

* Sets → No Duplicates
since hashtables store object, by their contents
It is very helpful for us

① HashSet

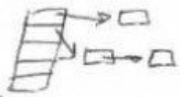
* In a hashtable the elements position is calculated by a hash function of its contents

* Hash tables obtain an Index from the hashcode by taking the remainder after division by the table length

[* The collections framework use bit masking rather than division → the pattern at the low end of the hashcode is significant and used to calculate the hashcode.
→ Multiplying by Primes will not shift Information from the low end → same as multiplying by a power of 2]

* Hash tables will eventually run out of storage space, we will have duplicates then → same hashkey but different values

Solution
→ A good hash function that spreads the elements out equally and when collisions occur new items are stored in a linked list.



→ added cost is following the chain call reference

→ As long as there are no collisions the cost at inserting or retrieving an element is constant

→ when the hashtable starts to fill collisions become more likely the probability of collisions is proportional to its load

→ Hashtable load = Number of elements / capacity (no. of buckets)

→ Collision occurs → Create linkedList → extra cost proportional to the number of elements in the list.

→ If the hashtable size is fixed performance will worsen

Solution → Increase table size by re-hashing → copying to a new and larger table when reaching the load factor

[Graphs]

* A tree, but a node can have more than one parent, cycles are allowed
- Undirected Vs. Directed (two way streets → undirected)
- Weighted Vs. Unweighted (Vertex is assigned a value or not)
- Cyclic Vs. Acyclic (cycles or no cycles)
- Embedded Vs. Topological (embedded → assigned geometric positions)

- Representation :

- Adjacency Matrix $n \times n$ matrix

  $M[i,j] = 1$ is (i,j) is and edge, 0 if it isn't

  → Manhattan Street Map

  15 avenues → crossing roughly 200 streets → 3,000 vertices
  6,000 edges

  3,000 X 3,000 cells → inefficient

  slow to add or remove / matrix must be resized / copied

- Adjacency lists in Lists

  → Linked lists to store neighbors adjacent to each vertex

  searching is hard we need to iterate

  → need for pointers When removing need to find all vertices

- Adjacency List in Matrix

  → eliminate the need for pointers

  → Keep a count of how many elements there are

[Heaps]

* A binary tree is a heap IFF
  → the key in the root is larger than that in children and both subtrees have the same property.

+ Heaps → Priority Queue
+ A complete tree is filled from the left.
+ When removing from the heap → take right most element and put it in the root, → maintain the property

→ Getting the Max value / Removing from heap is $O(n)$ height or $O(\log n)$
→ Addition (add to leaf, then move up) $O(n)$ or $O(\log n)$
→ used for Graph traversal.

Operation	Binary	Binomial	Pairing	Fibonacci ✓
findMin	$O(1)$	$O(\log n)$ or $O(1)$	$O(1)$	$O(1)$
deleteMin	$O(\log n)$	$O(\log n)$	$O(\log n)$	$O(1)$
insert	$O(\log n)$	$O(\log n)$	$O(1)$	$O(\log n)$
decreaseKey	$O(\log n)$	$O(\log n)$	$O(\log n)$	$O(1)$
merge	$O(n)$	$O(\log n)$ size of larger heap	$O(\emptyset)$	$O(1)$

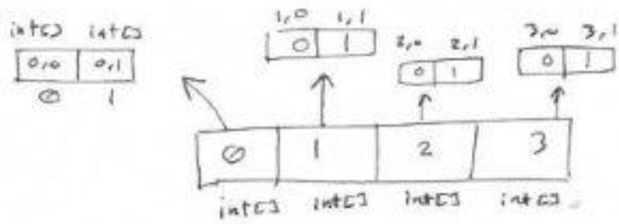
→ binomial heap is a collection of binomial trees
binomial tree with order 0 → 1 element
" " " " k → root which children are roots of binomial trees of k-1, k-2 ... 0

→ Fib Heap is a collection of trees satisfying the Min-heap property
the key of the child is greater or equal to parent

* Multi dimensional Arrays:

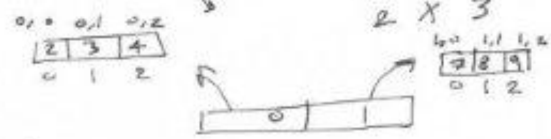
```
int [][] a = new int [4][2]
```

→ 4X2 Array → one Big Array that has four elements
each array element has two spaces



int[][] x = ~~new~~ { {2,3,4}, {7,8,9} }

2 X 3




```
int tri[][];
for (int r=0; r< tri.length; r++) {
    for (int c=0; c< tri[r].length; c++) {
        sout (tri[r][c]);
```

* Collections And Data Structures :

* Arrays : Implemented directly in hardware → properties of random access memory
  → Fast for accessing elements by position and iterating
  → Slow for Inserting and removing at arbitrary locations as that may require adjusting the position of other elements.

* Linked Lists : reference to next (Sometimes) previous elements
  → slow in accessing elements by position
  I have to follow the reference chain from the start
  → Insertion & removal are fast [constant time] by re- arranging the cell references.

* Hash Tables : Indexing elements based on their CONTENT
  → No support for accessing elements by position
  → Access, insertion, removal → very fast

* Trees → Organise elements by content + Store and retrieve in order they can (optional)

Algorithms use → time
→ space

* the space used for collections is usually proportional to the size of the collection.
* Variation in time requirements.

O notation : a way of describing the performance of an algorithm in an abstract way. It gives a way of describing how the execution time for an algorithm depends on the size of its dataset.
→ given that the data set is Large enough.

IMP

		Effect on the running time if N is doubled	ex:
$O(1)$	Constant	Unchanged	Insertion into hashtable
$O(\log N)$	logarithmic	Increased by constant amount	Insertion into tree
$O(N)$	Linear	Doubled	Linear Search
$O(N \log N)$		Doubled + amount proportional to N	
$O(N^2)$	Quadratic	Increased fourfold	Bubble Sort

	Add	Contains	Next
Hashset	$O(1)$	$O(1)$	$O(h/n)$ h: table capacity
Linked Hashset	$O(1)$	$O(1)$	$O(1)$
CopyOnWriteArraySet	$O(n)$	$O(n)$ linear search	$O(1)$
TreeSet	$O(\log n)$	$O(\log n)$	$O(\log n)$
ConcurrentSkipListSet	$O(\log n)$	$O(\log n)$	$O(1)$

	offer	Peek	Poll	size
Priority Queue	$O(\log n)$	$O(1)$	$O(\log n)$	$O(1)$
Concurrent Link Queue	$O(1)$	$O(1)$	$O(1)$	$O(n)$
Array Blocking Queue	$O(1)$	$O(1)$	$O(1)$	$O(1)$
Linked Blocking Queue	$O(1)$	$O(1)$	$O(1)$	$O(1)$
Priority Blocking Queue	$O(\log n)$	$O(1)$	$O(\log n)$	$O(1)$
Delay Queue	$O(\log n)$	$O(1)$	$O(\log n)$	$O(1)$
Linked List	$O(1)$	$O(1)$	$O(1)$	$O(1)$
Array Dequeu	$O(1)$	$O(1)$	$O(1)$	$O(1)$
Linked Blocking Queue	$O(1)$	$O(1)$	$O(1)$	$O(1)$

	get	add	contains	next	remove
Array List	$O(1)$	$O(1)$	$O(n)$	$O(1)$	$O(n)$
Linked List	$O(n)$	$O(1)$	$O(n)$	$O(1)$	$O(1)$

	get	contains	next
HashMap	$O(1)$	$O(1)$	$O(h/n)$
LinkedHashMap	$O(1)$	$O(1)$	$O(1)$
IdentityHashMap	$O(1)$	$O(1)$	$O(h/n)$
TreeMap	$\cancel{O(1)}$ $O(\log n)$	$\cancel{O(1)}$ $O(\log n)$	$\cancel{O(h/n)}$ $O(\log n)$
concurrentHashMap	$O(1)$	$O(1)$	$O(h/n)$

[Strings]

* All non-printable chars have either the First three bits as zeros or all seven lowest bits as one.
  → makes it easy to eliminate them before displaying Junk.

* Both the upper case and lower ones and the numerical digits appear sequentially → we can iterate through all letters / digits by looping from the value of the first symbol say "a" to last say "z"

* We can convert a char, say "I" to its rank by subtracting the first symbol.

* We can convert a char say "C" from upper to lower case by adding the difference of the upper and lower case starting character "C" − "A" + "a"

* a char is upper case Iff it is between "A" and "Z"

NewLine → ASCII = 10   Carriage return → ASCII = 13

[Combinatorics]

* I have 5 shirts and 4 pants → 5 × 4 = 20 ways to get dressed
* " " " " and the dryer ruined one of them
  → 5 + 4 = 9 possible ruined items

* $|A \cup B| = |A| + |B| - |A \cap B|$

  $|A \cup B \cup C| = |A| + |B| + |C| - |A \cap B| - |A \cap C| - |B \cap C| + |A \cap B \cap C|$

* Every Integer can be expressed in only one way as product of Primes $105 = 3 \times 5 \times 7$
* 2 is the only even, we verify the no. isn't even
* n is prime if it has no trivial factors below $\sqrt{n}$

NP Problem → at each step, I guess which possibility to try next there is no Information from previous attempts to determine what to try next.
→ investigate all possibilities

* The sum of two integers is even <u>Iff</u>

① they are both even

② they are both odd

→ go through and count evens and odds

no. of possibilities to combine K from a set of size N

no. of even / odd

$$= \frac{N!}{((N-K)! \cdot K!)} \quad K=2$$

* Prob of car passing intersection in 20 mins is 0.9

→ what is the prob of car passing in 5 mins window?

P (car passing in 20 mins) = 1 − (P no car passing in 20 min)

" " = 1 − (1 − (P car passing in 5 min) ^ 4)

" X = 1 − 10^(−0.25) = 0.4377

* A heuristic function is said to be admissible if it is no more than the lowest cost path to goal. Never overestimates the cost for reaching goal. (optimistic)

A^* complexity is polynomial → current Estimated Shortest path

Distance is calculated by $f(n) = g(n) + h(n)$

$g(n)$ total distance it has taken to get from starting point to current position

$h(n)$ estimated distance from current to the goal (destination)

Pseudo code:

1. Create list OPEN list with the starting node as its first element n_0
2. Create list Closed list which is initially empty
3. If open is empty exit
4. Take the first node of open, remove it, put it in closed, call it n
5. if n is the goal exit, trace the path from n to n_0
6. Expand node n, generate set M of its successors that are not already in open or closed.

   Add these members to OPEN.
7. Re-order the list OPEN in order of $f(n)$ values ascendingly.

(5),

$S \to A = 10.4 + 3 = 13.4$

$S \to D = 8.9 + 4 = 12.9$ ✓ $S \to D$

$D \to A = 10.4 + 5 = 15.4$

$D \to E = 6.9 + 2 = 8.9$ ✓ $S \to D \to E$

$E \to B = 6.7 + 5 = 11.7$

$E \to F = 3 + 4 = 7$ ✓ $S \xrightarrow{4} D \xrightarrow{2} E \xrightarrow{4} F \xrightarrow{3} G$ cost 13

A* guarantees this is optimal solution

* Branch and Bound (BnB):

- may not guarantee optimal solution

systematic enumeration of all candidate solutions, large subsets of fruitless candidates are discarded by using upper and lower estimated bounds.

looks for a bound which is guaranteed lower than the true cost.

cut (stop) search if cost + bound > best solution found.

* Distance to goal is always at least (0)

these algorithm returns a path as soon (the) a goal is reached, this is not always the shortest path

- Iterative Deepening (ID):

Starts with shallow depth (root), apply DFDB, if answer is not found increase the depth until an answer is found.

+ Finds shallow answers first.
+ Has always small frontier
→ Asymptotic guarantee
→ small overhead (additional work), worth for the advantages.

ID repeats work from depth 1 → d

$$\sum_{i=1}^{d} b^i = b^{d+1}/(b-1) \qquad \text{ratio to } b^d$$

$$b^{d+1} / b^d (b-1) = \underbrace{b/(b-1)}_{\text{times as much work as we need to}}$$

even if $b=2$, only twice as much work.

	Depth first	Breadth first	Depth first bounded	iterative deeping
complete	No	Yes	No (Yes if in bound)	Yes if in bound
Time	$O(b^d)$	$O(b^d)$	$O(b^d)$	$O(b^d)$
space	$O(bd)$	$O(b^d)$ proportional to the number of nodes at that level.	$O(bd)$	$O(bd)$
optimal	No	Yes if step costs are identical	No	Yes if step costs are all identical

b → branching Factor
d → depth

- Best First Search:

Expanding the most promising node. The node is estimated by a heuristic evaluation function $f(n)$

can be implemented using a priority queue. The most promising node is on top of the list

- Breadth first is poor except for very easy problems

Depth first is not useful without loop checking, not good with long branches

Depth-bounded → how to choose depth?

Iterative deepning is OK still need good h (iteration can be more than 1)

+ A^*

Best first with a good heuristic to find the least cost path from a node to goal. guarantees the first solution is optimal. we can stop searching immediately.

used for path finding, graph traversal

- Starts with routes that most likely lead towards goal. It takes into account the distance already traveled. (from the start not the local cost from the previous node only).

* ArrayLists saves Objects as instance of class object
→ When I want to retrieve an element back I should cast it back
  Dog d = (Dog) Arraylist. get (1);

* Wrapper Classes

```
int i = 25;
Integer wrapper = new Integer (i);
int z = wrapper. intValue ();
```

```
FileWriter writer;
writer. write ("Filename");
writer. close;
```
→ write to a file

* Format String

String. format ("%,.2F", 4123.456);
variable

separate every three digits with ,
approximate float to two digits precision → 4,123.46

format ("%,6.1f", 1142.00) → 1142.0
width = 6

* Data / time in Java   Calendar cal = Calendar. getInstance ();

* Object Serialization

① To a file:
```
FileOutputStream FS = new FileOutputStream ("Filename");
ObjectOutputStream OS = new Ob.... (FS);
OS. writeObject;
OS. close();
```
reverse to deserialize

② To A byte Array
* IMP Class Implements Serializable
```
byte [] byteBuffer = null;
ByteArrayOutputStream baos = new Byte..... ;
ObjectOutputStream oos = new ObjectO.... (baos);
oos. writeobject (o);
oos. close;
byte buffer = baos. toByteArray ();
```
I write to buffer then save from the buffer to byte []

De-Serialize
```
ByteArrayInputStream bais = new ByteArrayInputStream (byte[]);
objectInputStream is = new objectInputStream (bais);
object obj = is. read Object();
```
↳ casting.

* If a class has members that cannot be serialized
→ Define these members as transient