Data Structure and Algorithm

Data Structure: the organization of data in a computer's memory or in a disk file

Algorithm: a procedure for carrying out a particular task

Data base: a unit of data storage comprising many similar records

Basic Array:

Search: for each element: if search_value == element break;

Delete: for each element if search_value == element break; for the jth index of the element to end Array[j] = Array[j+1] size--

Insert Array[length] = Value length

Binary Search in ordered list: (Compare to the array: insertion & search??)

```
public int find(int value){ <Search in sort list>
  int upper, lower, current
  upper = length-1;
  lower = current = 0
  while(true){
    current = (upper+lower)/2
    if( Array[current] == value) Return current
    else if ( lower > upper) Return length
    else{
      if Array[current] < value
        lower = current+1
      else
        upper = current-1 }
  } // End while
}
```

```
insert(value){
  for(j=0; j<length;j++){
    if(a[j] > value) break}
  for(k= length k>j; k--){
    Array[k]= Array[k-1]}
  Array[j] = value
  nElement++
}
delete(value){
  int j= find(value)
  if j== length return faslse
  forlint j< nElement; j++){
    Array[j] = Array[j+1]}
  nElement--;
  return true;}
```

Invarient: conditions that remain unchanged as the algorrim proceeds

Abstract Data Types (ADT)

Sorted Linked List: Insertion & Deletion an arbitary item require O(N) comparision while for the minium value, it requires O(1) time

Iterator: Objects containing references to items in data structure, used to traverse data structure

Bubble sort (O(n²)) <base on principle> swap, comparision high

```
public sort(){
  for ( int i= Element-1; i>1; i--){
    for ( int j=0; j<i; j++){
      if Array[j] > Array[j+1]
        swap ( Array(j), Array[j+1])}
  }
}
```

Selection sort (O(n²)) <minize the swap> comparision still high

```
public selectionSort(){
  for ( int i= 0; i< length-1; i++){
    int min_index = i
    for ( int j=i+1 j< length; j++){
      if ( Array[min_index] > Array[length]){
        min_index = length
    } }
    swap ( Array[min_index], Array[j])
  }
}
```

Insertion Sort (O(n²)) better than bubble & Selection. <better> can change, flexible

```
public insertionsort() { // my self
  cur_index = 1
  While ( cur_index< array.length)
    for(int i=0; i< cur_index; i++)
      if( array[i] > array[cur_index]{
        swap i, cur_index;
      }
    } end
    cur_index++;
```

```
for(j=1; j< num.length; j++){
  key= num[j]
  for( i=j-1; (i>=0 && a[i]>key); i--){
    a[i+1] = a[i] }
  a[i+1] = key
```

QuickSort() (O(nlogn))

```
rec QuickSort( int low, int high){
  if (high<= low) return;
  int privot = A[high]
  int index = recQuickSort( low, high, privot)
  recQuickSort( low, index-1)
  recQuickSort( index+1, high)
```

```
QuickSort(int low, int. high, int privot){
  int left= low-1;
  int right = high
  while(true){ while ( left< high && A[++left]<privot){}
    while ( right> low && A[--right]> privot){}
    if (left >= right) break
    else{swap ( left, right}
  } swap( left, high)
    return left
```

May behavior bad when choose not even privot & divide too many subarrays need (O(n²) if reverse sorted array)

Merge Sort (N logN) twice of space than simple arry

```
recMerge :: if low== high return
else mid = (low+high)/2
rec Merge( Array, low, mid)
rec Merge( Array, mid+1, high)
Merge ( Array, low, mid, high):: create butter
  same size as Array
  int lindex = low
  int rindex = mid
  while( lindex<=mid-1
    && rindex <= high){
    if array(lindex)>array(rindex)
      workspace[i++]= array[rindex]
    else
      workspace[i++]= array[lindex]
  While(lindex <= mid-1)
    workspace[i++]= array[lindex++]
  While( rindex<= high)
    workspace[i++]= array[rindex++]
  for(int j=0 j<high-low+1 j++)
    array[low+j] = workspace[j]
```

Stack: Last in First out (O(1)) can be used to reverse word, check syntax

Queue: First in First Out (O(1)) O(1)

Poriority-queue implementation: Search, insert: O(N)

Single Linked List: class Link { data, Link}

Double linked List: class DLink { data, Link, Link}

Link List: Insertion & Delection & Finding a specific items need searching through O(N) But No need to shift element compare to the array. use exact memory as it needs

S2.compareTo(S1)

S1<S2 <0

S1>S2 >0

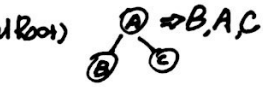
```
Shell sort(){ int h=1;
  while( h<= nElement/3)
    h= h*3+1;
  while(h>0){
    for ( i=h; i< nElement i++)
      temp= A[i]
      j = i
      while ( j>h-1 && A[j-h]
  >= temp){ A[j]= A[j-h]
        j -= h; }
      A[j] = temp } // for
    h=(h-1)/3; } // while
```

Binary Tree: search a tree quickly, as you can an ordered array, and you can also insert and delete items quicky. as you can with a link list

```
class Node {
    int value
    Node leftChild
    Node rightChild }
public find() {
  Node cur = root;
  While (cur.value != key){
   if (cur.value > key){
      cur = cur.leftchild;
   else { cur=cur.rightChild}
   if (cur == null){
      return null}
   }
  return cur
```

```
public insert(){ int, value){
   Node n = new Node(value)
   if(root == null){ root=n }
   else{ Node current = root;
        Node parent;
        While (true) {
        parent = current
        if ( value < current.value){
            current = current.leftChild
            if (current == null){
                parent.leftchild = n
                return; }
        else { current = current.rightChild
            if ( current == null){
                parent.rightChild = n
                return; }
        }} // end while, end else
}
```

```
public void inOrder(node localRoot)
{ if ( localRoot != null) {
     inorder (localRoot.Leftchild)
     localRoot.display()
     inorder (localRoot.Rightchild)}}.
```



=> B, A, C

```
public Node minimum(){
     Node current, last;
     current = root:
     While ( current ) = null)
     { last= current
       current = current.leftchild;
     return last;}
```

```
public boolean Delect (int key)
  Node current, parent = root
  boolean isleft = true;
  While (current.data != key){
     parent=current
     if ( key < current.data){
          isLeft = true
          current = current.leftchild}
     else { isLeft = false
          current = current.rightchild}
  } // After finding the key// if(current == null){ return false}
  ① if ( current.leftchild == null && current.rightchild == null)
     { if (current == root){ root = null }
       else if (isLeft){ parent.leftchild = null }
       else parent.rightchild = null }
  ② else if (current.rightChild == null){
       if (current == root){ root = current.leftchild }
       else if (isleft){ parent.leftchild = current.leftchild }
       else { parent.rightchild = current.leftchild }
  ③ else if ( current.leftchild == null){
       if (current == root) { root = current.rightchild
       else if (isleft){ parent.leftchild = current.rightchild
       else { parent.rightchild = current.rightchild}
  for the third situation need to find successor
     else { Node successor = findSuccess(current)
       if (current == root) root = successor
       else if (isleft) parent.leftchild = successor
       else parent.rightchild = successor
       successor.leftchild = current.leftchild
  return true
```

① node to be delete is leaf
② node to be delete has one child
③ node to be deleted has two child

find key
record parent
record position (right/left)

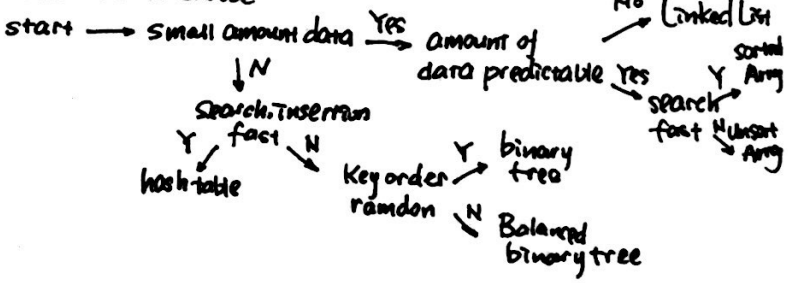
```
findSuccessor (Node delNode)
{ Node suParent, su = delNode
  Node current = delNode.rightChild
  While ( current != null)
  { suParent = current su
    su = current
    current = current.leftchild;
  if( su != delNode.rightchild ){
  { successorParent.leftChild = successor.rightChild
    Successor.rightChild = delNode.rightChild}
  return successor;
```

Hashtable: very fast insertion and searching but based on array. so difficult to expand performance degraded if tables becomes too full No way to visit items in some order

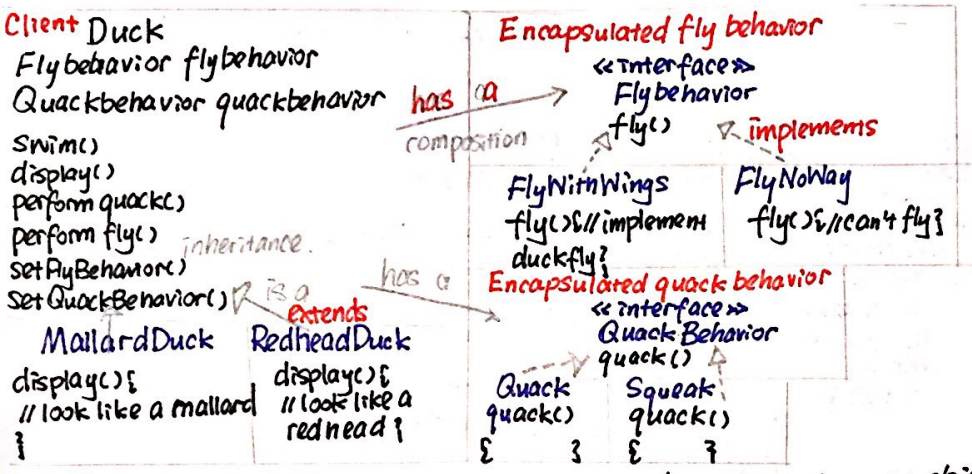
```
public DataItem find (int key)
{ int hashIndex = hashfunction(key);
While(hasharray[hashIndex] != null){
  if hasharray[hashIndex] == value
     return hasharray[hashindex]
  ++hashindex;
   hashindex %= hashsize}
  return null}
```

```
Insert(Item item)
 int key = item.key
 int hashindex = hashFun(key);
While ( hashArray[hashindex] != null && hashArray[hashindex].value != -1
{   hashindex++;
    hashindex %= arraysize }
hashArray[hashindex] = item.}
public DataItem delete (int key)
 int hashindex = hashfun(key)
While ( hassarray[hashindex] != null {
    if (hasharray[hashindex].value == key){
        Item temp = hasharray[hashindex]
        hasharray[hashindex] = null
        return temp}
    hashindex++
    hashindex %= arraysize}// end while
    return null}
```

Which one to choose



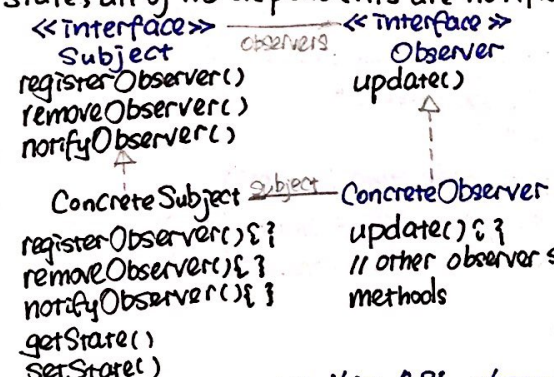
The Strategy Pattern: defines a family of algorithms, encapsulates each one, and makes them interchange. Strategy lets the algorithm vary independently from clients and not use it.




```
public NewTypeDuck(){
    flybehavior = new NoWayFly()
    quackbehavior = new Quack()

public static void Main(String[] args)
{   Duck model = new NewTypeDuck();
    model.performFly()
}
```

The Observer Pattern: defines a one-to-many dependency between objects so that when one object change state, all of its dependents are notified and updated automatically



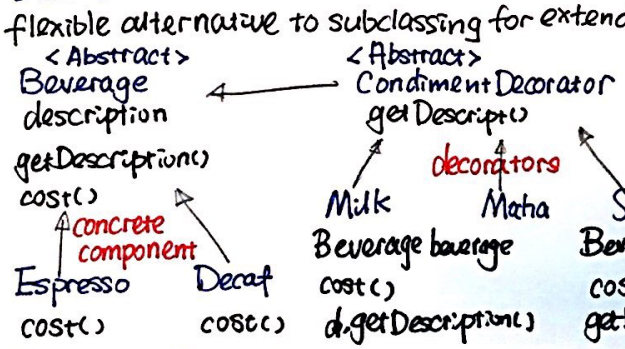
loose coupling: they can interact, but have very little knowledge of each other

```
Weather Data implements Subject
public WeatherData(){observers = new ArrayList()}
public notifyObservers(){
  for(int i=0; i< observers.size(); i++){
      Observer observer = (Observer)observers.get(i)
      observer.update(temp, humidity, pressure)}
public void measurmentChanged(){ notifyObservers}
CurrentConditionDisplay implements Observer, DisplayElement
public CurrentConditionDisplay(Subject WeatherData){ this.WeatherData = WeatherData
  weatherData.registerObserver(this);}
public void update(float temp, float humidity, float pressure){ display(temp)}
```

In Java, there is bilud in API. observable (subject) and observer ('observer) setchange() must call before notifyOb

```
import java.util.Observable; import java.util.Observer;
public class WeatherData extends Observable{ private float temp, humid, pressure; public WeatherData(){}
public void measurementChanged(){ setchanged(); notifyObservers();}
public class CurrentConditionDisplay implements observer { Observable observable; public CurrentConditionDisplay(
Observable observable){ this.observerble = observable; observable.addObserver(this);} public update(Observable, obs
Object arg){ if(obs instanceof WeatherData){ WeatherData weatherData = (WeatherData) obs; display();}
```

Decorator Pattern: D. attaches additional responsibilities to an object dynamically. Decorators provide a flexible alternative to subclassing for extending functionality.



Decorator class mirror the type of the components they decorate. (same class)

```
public class Beverage{ String description="Unknown Beverage"
abstract
    public String getDescription(){
        return "Unk description;}
    public abstract double cost();
public abstract class CondimentDecorator extends Beverage
    public abstract String getDescription();
public class Espresso extends Beverage{
    public Espresso(){ description = "Espresso"
    public double cost(){ return 1.99;}
public class Mike extends CondimentDecorator
    public Mike(Beverage beverage){this.beverage = beverage
    public getDescription(){ return this.beverage.getDes
        + "Mike";}
    public cost(){ return 0.2 + this.beverage.cost()}
public static void main(String args[]){
    Beverage beverage = new Espresso
    beverage = new Mike(beverage);
    beverage = new Mike(beverage);
```

Factory Method Pattern: Defines an interface for creating an object, but lets subclasses decided which class to instantiate. Factory Method lets a class defer instantiation to subclasses.

Pizza (abstract) — Product Class

NYstyleCheesePizza, NYstyleClamPizza, NYstyleVeggiePizza, ChicagoStyleCheesePizza, ChicagostylePepperPizza, ChicagoStyleVeggiePizza

PizzaStore (abstract) — CreateClass

createPizza()
orderPizza()

NYPizzaStore, ChicagoPizzaStore

createPizza()

((only concrete creator is responsible for creating one or more concrete product))

```
public abstract class PizzaStore {
    public Pizza orderPizza(String type){
        Pizza pizza;
        pizza = creatPizza(type);
        pizza.prepare()
        pizza.bake()
        pizza.cut()
        pizza.box()
        return pizza;}
    protected abstract creatPizza(String type);}
```

```
public class NYPizzaStore extends PizzaStore{
    Pizza CreatePizza(String item){
        if(item.equals("cheese")){
            return new NYStyleCheesePizza();}
        else if(item.equals("veggie"){
            return new NYStyleVeggiePizza();}
        else return null;
```

```
public abstract class Pizza {
    String name;
    String dough;
    String sauce;
    void prepare(){ //implementation}
    void bake(){ //implementation}
    void cut(){ //..}
    void box(){ //..}}
```

```
public class ChicagoStyleCheesePizza extends Pizza
    public ChicagoStyleCheesePizza(){
        name="chicago cheese"
        dough=".."
    void cut(){override func if want}
```

```
main(){ PizzaStore pizzaStore = new NYPizzaStore()
        Pizza pizza = pizzaStore.orderPizza("cheese")
```

Thread Prep

Critical Section: part of multi-threading program may not be concurrently executed by more than one process.

Mutual Exclusion: <<synchroized>> a property of concurrency control, for the purpose of preventing race condition by preventing one thread of execution never entered its critical section at the same time that other concurrent thread of execution enters its own critical sections.

Each thread has its own stack, share one heap. Stack: call stack, local. parameter, heap: object.

Deadlock: two threads need resources from each other.

Livelock: two threads need / keep waiting other to finish.

| No pre-emption => have priority
| Mutual exclusion => Hold resource Wait other
| Circular wait: => lock in order
Hold & Wait: =>

join(): wait for another thread and start execution once the thread has completed

Java Core:

JVM / JRE: implementation of JVM / JDE: JRE + development tools

compiler → byte code → machince code
(source code) ~compile JVM.

Immutable: can not be modified once created.

String buffer: safe String builder: not

cluster / noncluster index: physical store vs. map to the physical

Hashtable is synchronized byt hashmap is not, hashmap allows null while hashtable does not.

keys => . hash(key) => find buckets (index) => . entries(may be in linked list))

When implement equals must implement hashcode

DPatten & Database Review.

MySQL & SQL Server: relational database management system use SQL

DDL: CREATE ALTER DROP DML: ... SELECT INSERT UPDATE DELETE
definition manipulation

BETWEEN: range IN: a set of values LIKE: a pattern

Subquery: Select statement embedded in clause of another SELECT.

In-line view: FROM [SELECT statement]

Collection, Set, List, Map
Set: no duplicate no order
List: order
Map: key-value pair

Tree: ordering is important
Hash: speed

Link-List: adding a node is quick, irrelevant of where you're adding
conversely node is relative slow O(n)
addition / remove very quickly.

ArrayList: adding, reove all backwards, remove, move frontwards
dynamical allow a new array
fast read access, no chang significantly

Hashmap: array of reference (entry table default:16) brackets linklist
where hash entries stores, load factor. 0.75
if greater, rehashing, double size

Database SQL Interview Question

Relational Database management System (RDBMS) vs. Database management System (DBMS):
RDBMS stores data in tabular form, while DBMS stores data as files /No relationship between files or tables in DBMS.

Primary key: one or combination of rows uniquely specify a row [Unique. Not null]

Foreign key: a field in one table that uniquely identified a row of another table.

Normalization: is a process of minimizing redundancy and dependency by organizing fields and table of a database

Index: tuning method of allowing faster retrieval of records from the table

Unique index: can be apply automatically when primary key is define.

Cursor: a control which enables traversel over the rows or record in the table.
a pointer to one row in a set of rows.

Stored Procedure: function consists of many SQL statement to access the database system

Trigger: automatically execute with response to some event on a table or view