Data Structures - MidTerm 2025 (Melody N.)

Data Structure - a particular implementation for storing & managing data.

Chp 1: Abstract Data Types – Set of values & operations, not necessarily a collection, independent of implementation.

Ex. Bag, List (built in Python DS), Stack, Queue, Priority Queue (Special kind of bag)

Collection - general group of objects vs. **ADT** - Formal model defining behavior (operations + constraints)

Induction - Prove solution up to n finite natural terms.

Sum of natural numbers: $n(n+1)/2$

1) Base: Prove P(K) true for K=1
2) Assumption: Let P(K) true for all K in N and K>1
3) Induction: Prove P(K+1) is true

Recursion – Function repeats itself again & again by calls itself directly/indirectly to move further. Stops at solution. (operation on a smaller subarray w/ each step)

1) Base condition
2) Recursive step: Divides big problem into small instances, function later combine results solve w/ recursion

Ex. Fibonacci $F_N = F_{N-1} + F_{N-2}$ where $F_0 = 0$

$F(n) = F(n-1) + F(n+2)$

Factorial: $N! = N * (N-1)!$

$C = 5(F-32)/9$

Object - represent real world entity, state - descriptive characteristics. behaviors what it can do (or can be done to it)

Primative Java Data Types (8)

Integers – Byte (8), short (16), int (32), long (64 bits), char

Floating Point - float, double

boolean

Classes - blueprint of an object → methods define behaviors

Interfaces – Describes what a data structure does (ADT) Defines select, insert, & update statements

ADT → ArrayBag

- -bag: T[] - finite collection of objects. No order. Same data type, duplicates allowed
- -numberOfEntries: integer same data types
- -DEFAULT_CAPACITY: integer empty bag ok

- +getCurrentSize(): integer O(1), works even if unordered
- +isEmpty(): boolean O(1)
- +add(newEntry: T): boolean O(1) ← core method
- +remove(): T O(1)
- +remove(anEntry: T): boolean O(1), O(n)
- +clear(): void O(n) No worries deallocating nodes after clear
- +getFrequencyOf(anEntry: T): integer# occurance O(n)
- +contains(anEntry: T): boolean O(1), O(n)
- +toArray(): T[] O(n)
- -isArrayFull(): boolean Bag full using linked implementation

Queue

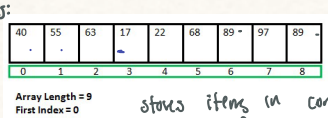
- A method can change state of object passed to it as argument

Operation Speeds:

✓ Fastest: Access, O(1) insert at end

✗ Slowest: Insert/Delete at beginning/middle O(n)

Array - (fixed-size)

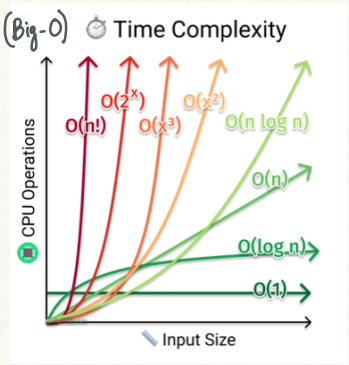


stores items in continuous memory, calculable addresses, faster access to element, not efficient wastes more space than a chain

Chp 2: Algorithm Analysis

Efficient algorithms → reduce run time ✓ (complexity)

Optimal performance Goal: O(1) or O(log n)



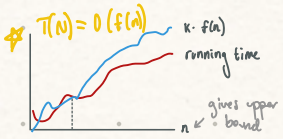
Function		Operation:
efficient	C	Constant – Input doesn't matter
	Log N	Logarithmic – Ex. Binary search
	$Log^2 N$	Log-squared – Merge/quick Sort, Binary Search
moderate	N	Linear
	N log N	N log N
slower	N^2	Quadratic
	N^3	Cubic
slow	2^N	Exponential – Doubles w + input
	n!	Factorial

Time Complexity - implemenation, distribution of data & inputs used. Provides upper bound on run time, use to compare efficiency of different algorithms, no matter input size more important than

(order of magnitude)

Big O(n) notation – running time grows with input size. Ex. Size (N) doubles, time doubles.

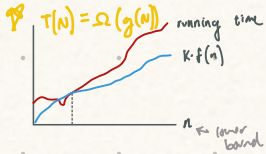
1) Ignore constants.. Ex. $5n \rightarrow O(n)$ constant 5 is insignificant to growth
2) Dominant terms Alg. w/ multiple terms $n^2 + 3n + 4 \rightarrow O(n^2)$



If positive constants C and N0

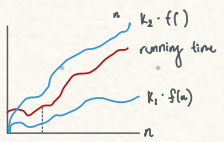
Big Oh (O) function that describes 'upper bound' aka worst case fastest algorithm runs,

$T(N) \leq cf(N)$ for all $N \geq n0$



Big Omega (Ω) - function describes 'lower bound' aka best case

$T(N) \geq cg(n)$ for all $N \geq n0$



$T(N) = \Theta(h(N))$ if & only if

$T(N) = O(h(N))$

$T(N) = \Omega(h(N))$

Big Theta (Θ) - function describes 'tight bound' aka when Big O & Ω are same type func Lower bound, smaller constant k_1 Upper bound, larger constant k_2

Running Time Calculations

input size	required time complexity
$n \leq 10$	$O(n!)$
$n \leq 20$	$O(2^n)$
$n \leq 500$	$O(n^3)$
$n \leq 5000$	$O(n^2)$
$n \leq 10^6$	$O(n \log n)$ or $O(n)$
n is large	$O(1)$ or $O(\log n)$

Sorted otherwise $O(n^2)$ Fast →

Ex. O(n) algorithm 1,2,3,...n O(n)

```
long sum = 0;
for (long j = 1; i <= n; i++)
  sum = sum + i
```

$O(n^2)$ algorithm

```
sum = 0;
for (long j = 1; i <= n; i++)
{
  for (long j = 1; j <= i; j++)
  sum = sum + 1;
}
```

1 2 3 ... n

$\therefore O(n^2)$

Algorithm Efficacy (Big-O Notation)

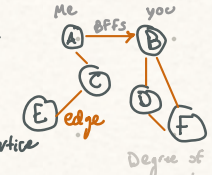
Algorithm Type	Best Case	Worst Case	Stable?
Constant Time	O(1)	O(1)	
Linear Search	O(1)	O(n)	
Binary Search	O(1)	O(log n)	
Bubble Sort	O(n)	$O(n^2)$	✓
Merge Sort	O(n log n)	O(n log n)	✓
Quick Sort	O(n log n)	$O(n^2)$	✗
Insertion Sort	O(n)	$O(n^2)$	✓
Heap Sort	O(n log n)	O(n log n)	✗
Selection Sort	$O(n^2)$	$O(n^2)$	✗

Stable Sorting: Maintains order of equal elements
Unstable Sorting: May change order of equal elements

Shortest path between vertices?

Graph: Collection of vertices (nodes) (people) & edges (connections) (relationships).

- Directed (Digraph): Edges only go one way. A→B
- Undirected: Edges have NO direction A↔B Types
- Weighted: Edges have weights (distances, costs)



Graph Representations:

- Adjacency Matrix: $O(V^2)$, faster lookup, high memory usage
- Adjacency List: $O(V+E)$, efficient for sparse graphs.

Degree of separation → weight

Graph Traversal:

- BFS (Breadth-First Search): Uses a queue (Shortest path in unweighted graphs)
- DFS (Depth-First Search): Uses a stack (or recursion)

Chp 3: Stacks, Lists, Queue - Linked Data Structures (dynamic)

Singly Linked List - linear, stores elements sequentially in nodes

Node structure: {data, Next} — reference to data of next node

Operations: Insert at head O(1) ✓ — Insert at tail O(n) — Delete from tail O(1) ✓ — Search O(n)

Less rigid storage, efficient insertion & delete (Create / Destroy, length, find)

extra memory pointers ✗

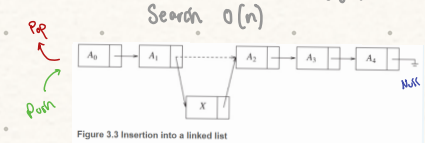


Figure 3.3 Insertion into a linked list

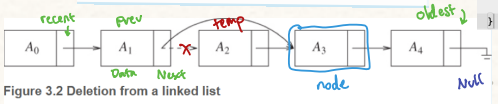


Figure 3.2 Deletion from a linked list

```
public static void swapWithNext( Node beforep )
{
  Node p, afterp;

  p = beforep.next;
  afterp = p.next;   // Both p and afterp
      assumed not null.

  p.next = afterp.next;
  beforep.next = afterp;
  afterp.next = p;
}
```

O(1) (Singly / doubly)

✓ Fast: Insert / delete at head

✗ Slow: Access, searching O(n) deleting last node (if singly)

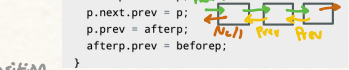
Doubly Linked List

Node Structure: {prev, data, next}

✓ Faster traversal (both directions), easy delete

✗ Extra pointers = more memory

```
public static void swapWithNext( Node p )
{
  Node beforep, afterp;

  beforep = p.prev;
  afterp = p.next;

  p.next = afterp.next;
  beforep.next = afterp;
  afterp.next = p;
  p.next.prev = p;
  p.prev = afterp;
  afterp.prev = beforep;
}
```



Sequential Search: linear data structure traversed until record's key found.

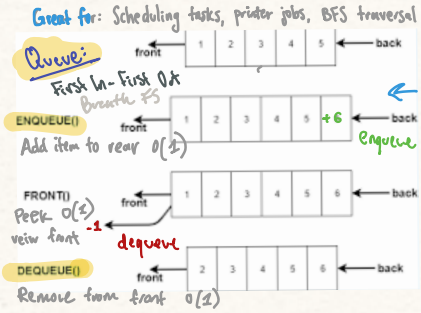
worst case, O(n) target value at last position

Ex. Depth First Search

Queue:

Great for: Scheduling tasks, printer jobs, BFS traversal

First In - First Out



Circular Queue: Uses modulo arithmetic to reuse space.

Deque (Double ended): Insert/remove from both ends.

Priority Queue: Items dequeued based on priority. (not order)

✓ insert/delete O(log n)

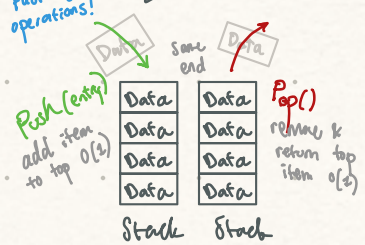
✗ Constant-time access

Stack:

Great for function calls, parsing, undo features (reverse), expressions

Fast O(1) operations!

Last in - First Out



Push(entry): add item to top O(1)

Pop(): remove & return top item O(1)

Array-based - Must resize if full O(n).

Linked-list - No resizing needed, O(1) all operations. Same type elements

Complexity Analysis

Operation	Stack	Operation	Array
push	O(1)	Insert	O(n)
pop	O(1)	Delete	O(n)
peek	O(1)	Access	O(1) ✓ indexing
isEmpty	O(1)	Traversal	O(n)

Add entry to top — Remove & return top — View top/front item w/o removing — Check if stack is empty

Postfix notation: (Reverse polish N)

Expressions w/o parenthesis, operators come after numbers (operands)

left → right (stack)

1 + 2 → 1 2 +

1 + 2 * 3 → 2 3 * 1 +

(1 + 2) * 3 → 1 2 + 3 *

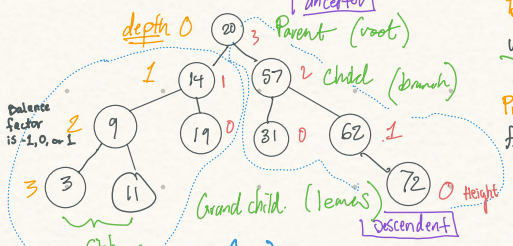
7 + (8 x 8^2 + 4) → 7 8 8 2 ^ 4 + +

Infix — Postfix

Chp 4: Trees, Sets, Maps

Avg Running time: non-recursive (non-repetitive) or O(N) — O(log N)

Binary Search Tree - Left subtree < Root < Right Subtree



Property 1: Each node up to two successor nodes

Property 2: unique path from root to every other node

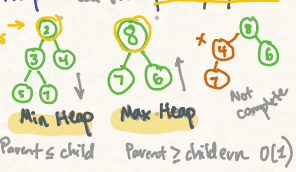
Search, Insert, Delete: O(log n) average, O(n) worst

Depth-first - go down branch before backtracking. stack

Breadth-first - Searches all children before moving deeper

AVL tree (Red Black) - height balanced tree → Maintain balance for O(log n) operations

Heap - used for priority queues



Parent ≤ child — Parent ≥ children O(1)

TreeSet

Set of unique elements vs. Tree Map key-value pairs

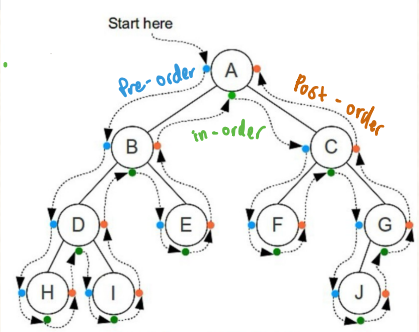
✓ better time complexity — ✓ Good for temp empty info

```
TreeSet<String> treeSet = new TreeSet<>();

// Add elements to the TreeSet
treeSet.add("apple");
treeSet.add("banana");
treeSet.add("orange");

// Print the TreeSet
System.out.println(treeSet);

// Remove an element from the TreeSet
treeSet.remove("banana");
```

Output: [apple, banana, orange] / [apple, orange]

```
TreeMap<Integer, String> treeMap = new TreeMap<>();

// add some key-value pairs to the TreeMap
treeMap.put(3, "three");
treeMap.put(1, "one");
treeMap.put(2, "two");

// print out the TreeMap
System.out.println(treeMap);

// get the value associated with a key
String value = treeMap.get(1);
System.out.println("The value associated with key 1 is: " + value);

// remove a key-value pair
treeMap.remove(2);
```

Output:
{1=one, 2=two, 3=three}
The value associated with key 1 is one
{1=one, 3=three}



Pre Order Traversal
```
def iterate(root):
    print(cur)
    iterate(root.left)
    iterate(root.right)
```
Root → Left → Right

In-order Traversal
```
def iterate(root):
    iterate(root.left)
    print(cur)
    iterate(root.right)
```
Left → Root → Right

Post-order Traversal
```
def iterate(root):
    iterate(root.left)
    iterate(root.right)
    print(cur)
```
Left → Right → Root

Level-order (BFS)
```
def iterate(root):
    q = [root]
    while q:
        c = q.pop(0)
        if c:
            print(cur)
            q.append(c.left)
            q.append(c.right)
```
Root
Level 1, Level 2, etc...

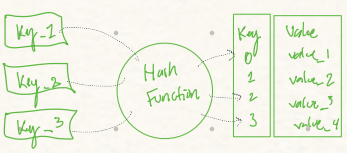
Chp 5: Hashing & Hash Tables

- Allocate a fixed size array n.
- When inserting, do math to search for location — O(1) target
- To delete, do search, then remove number
- When table gets too crowded, resize.

Hash Function: Maps keys to index values in an array

Set → HashSet - just stores items

dict → Hash map - stores key, value pairs. (integer)

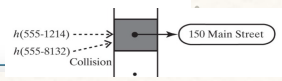



```
Algorithm add(key, value)
index = h(key)
hashTable[index] = value
```

```
Algorithm add(key, value)
index = h(key)
hashTable[index] = value
```

- **Most hash functions are not perfect,**
  - **Can allow more than one search key to map into a single index** (same hash code)
  - **Causes a collision in the hash table** Lookups → O(n)
- **Consider** `tableSize = 101`
- `getHashIndex(555-1214) = 52`
- `getHashIndex(555-8132) = 52`

also!!!



Collision Resolution Strategies:

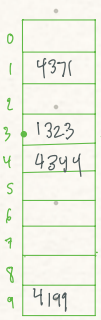
simple + efficient for small tables

Separate Chaining - Use linked lists at each index.

Given input {4371, 1323, 6173, 4199, 4344, 9679, 1989} and a hash function $h(x) = x \bmod 10$

4371 % 10 = 1 → slot

Hash table range 0-9



Look ups increase, memory + run time

Best for fast look ups & data set size unknown.

Same slot → collision → chain

✓ Fast: Insert, search, delete

✗ Worst case collisions

Linear Probing - If location occupied, check next open slot

Rehash(key) = (n + 1) % table_size

Quadratic Probing - Use squared increments to find a new slot.

Reduces clustering, but interval increases

1. Start at O.G. location
2. Only if occupied, check other slots

collision at $h(x) \% S$

try: $(h(x) + 1^2) \% S$

$(h(x) + 2^2) \% S$

$(h(x) + 3^2) \% S$ ...

Complexity:

Search - O(1) best case, O(n) worst case w/ collisions

Insert - O(1) best case, O(n) if table is full