Midterm Review

Recess Week, AY 19/20 Sem 2

Wang Zhi Jian
wzhijian@u.nus.edu

Midterm

Asymptotic Analysis

Sorting

Linked Lists, Stacks, Queues

Hashing

Big-O Notation

General Rules

1. Retain only the dominant term
   e.g. $n^4 + n^3 + n^2 = O(n^4)$

2. Ignore all coefficients
   e.g. $3n^2 = O(n^2)$

Iterative Functions

In general, count total number of iterations performed.

Beware of nested loops with varying number of iterations in the inner loops.

e.g. Tutorial 1, Question 2d.

```
while (n > 0) {
    for (int j = 0; j < n; j++)
        System.out.println("*");
    n = n / 2;
}
```

Recursive Functions

Use recursion tree. Follow the steps to draw and analyze a recursion tree.

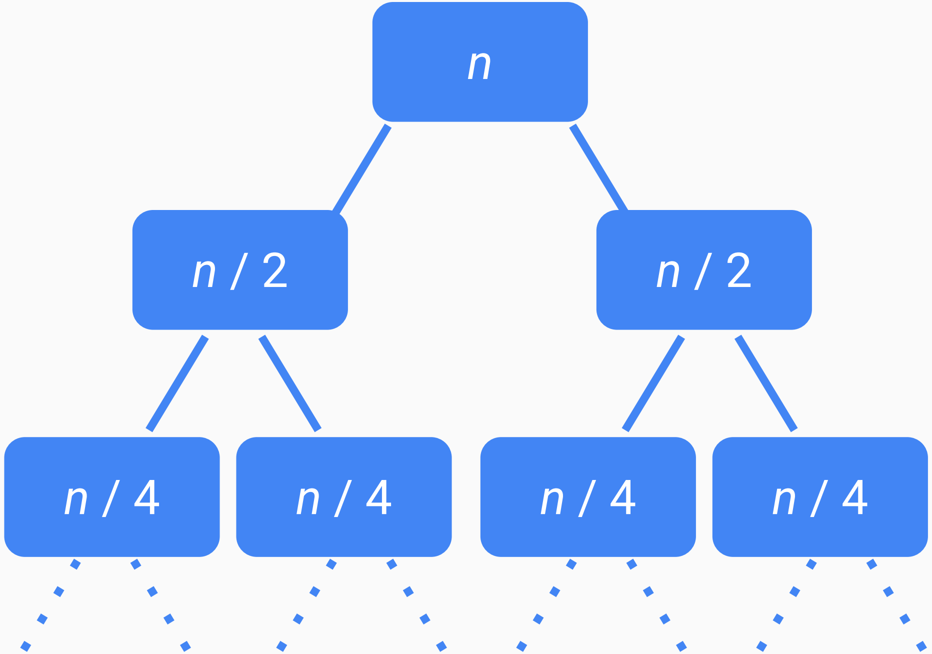
1. Draw the recursion tree out
2. Find the height of the tree
    a. The height of the tree usually corresponds to the **number of terms** you need to sum
3. Find the work done at every node in the recursion tree
4. Find the work done at every layer of the recursion
    a. This is just the sum of the work done by every node in each layer
    b. See you can spot some kind of pattern

Asymptotic Analysis

```
void foo(int n){
    if (n <= 1)
        return;
    doOhOne(); // doOhOne() runs in O(1) time
    foo(n/2);
    foo(n/2);
}
```

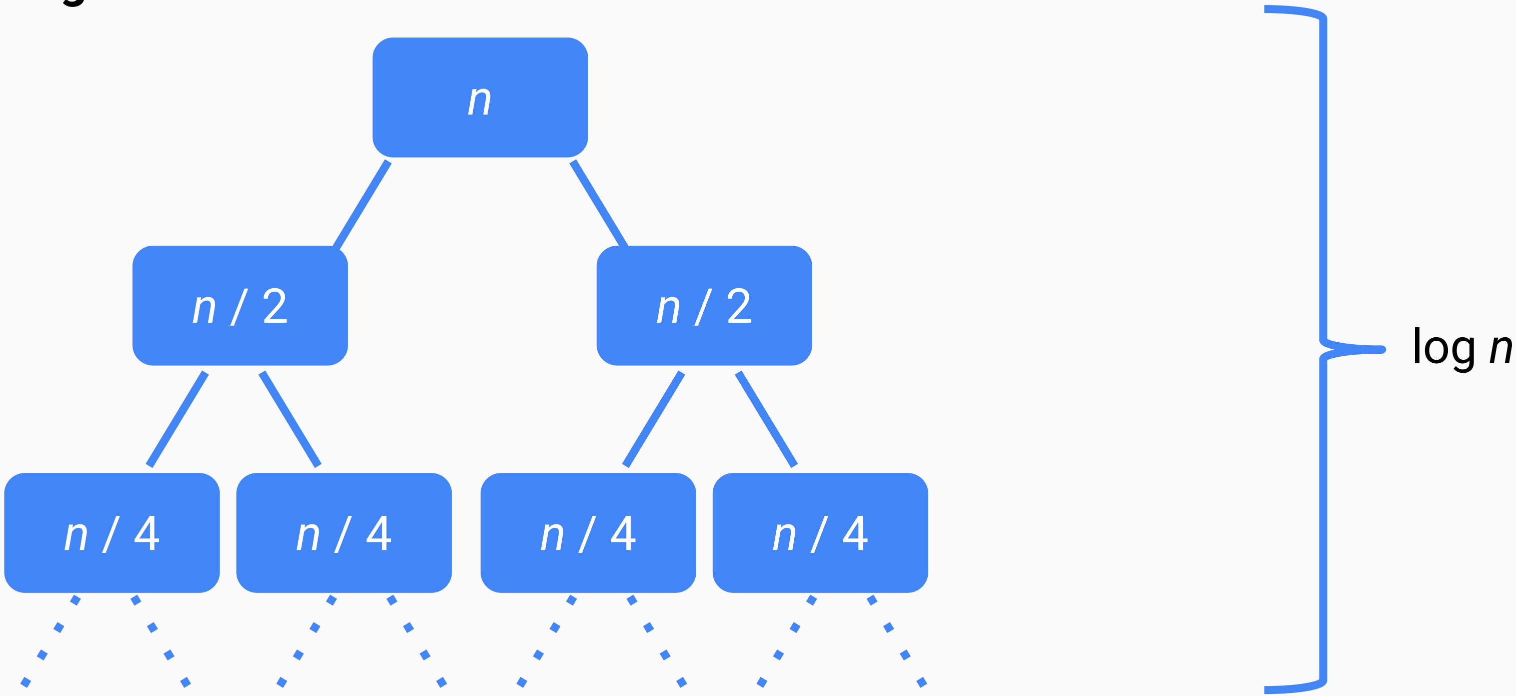
Question 2

Step 1: Draw the recursion tree

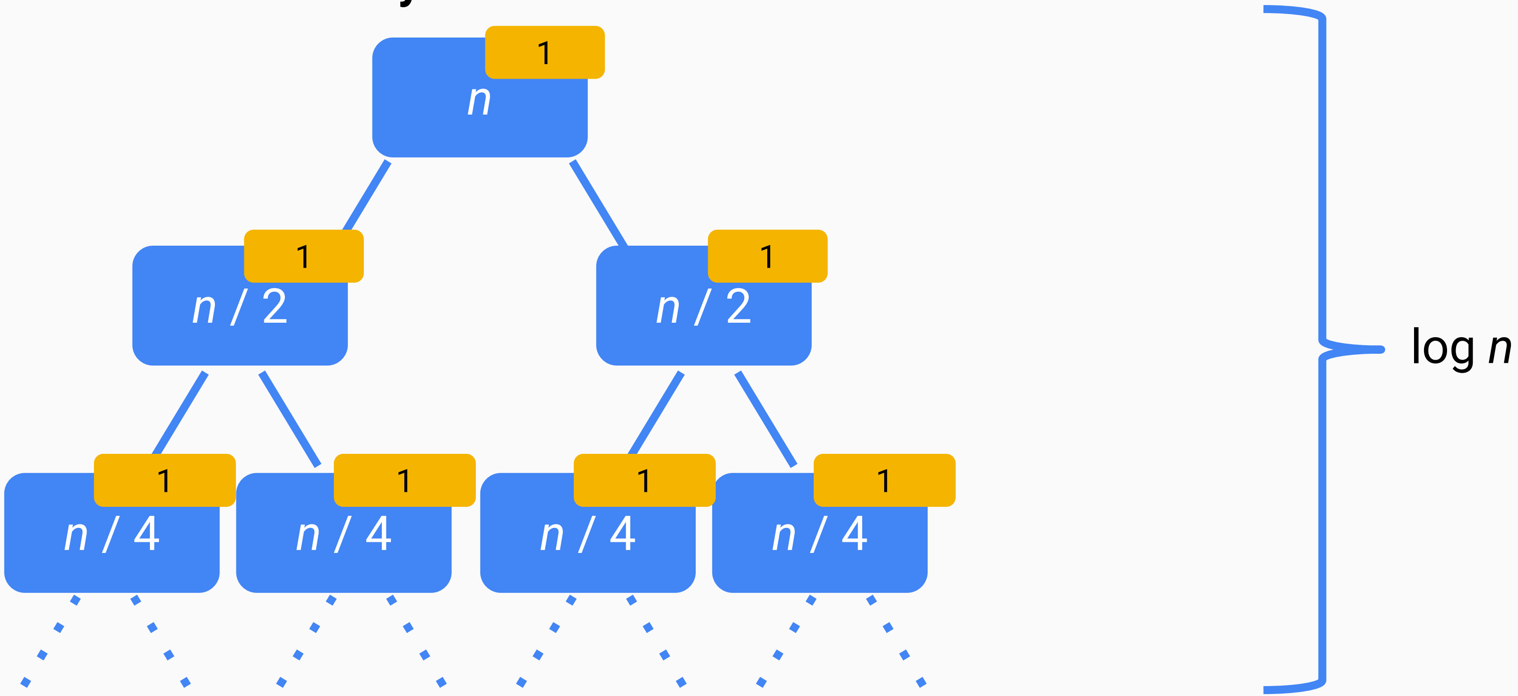


Question 2

Step 2: Find the height of the tree

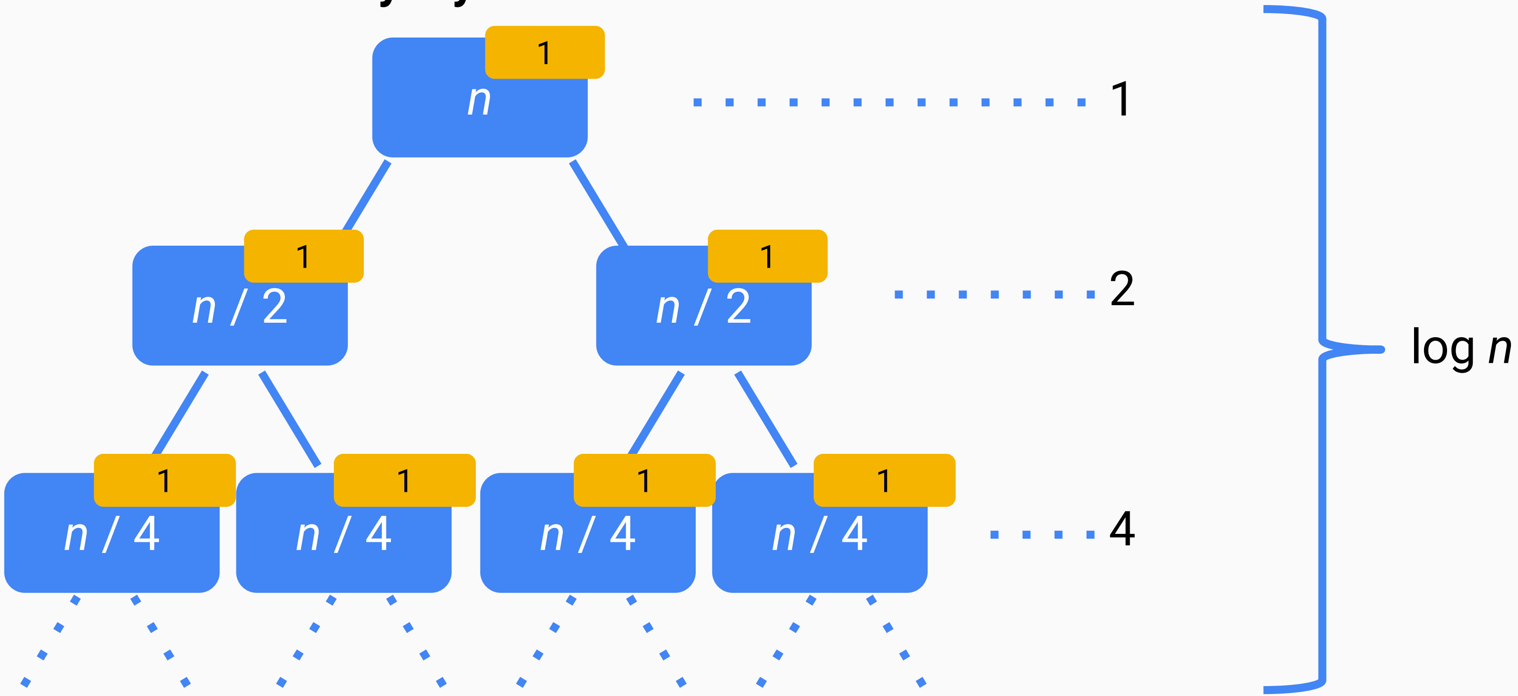


Step 3: Find the work done for every node



Question 2

Step 4: Find the work done at every layer



Question 2

Finding a Pattern

At the 1st layer of recursion, 1 operation is performed.

At the 2nd layer of recursion, 2 operations are performed.

At the 3rd layer of recursion, 4 operations are performed.

…

At the kth layer of recursion, 2^{k-1} operations are performed.

Question 2

Finding a Pattern

At the 1st layer of recursion, 1 operation is performed.

At the 2nd layer of recursion, 2 operations are performed.

At the 3rd layer of recursion, 4 operations are performed.

...

At the $\log n$ th layer of recursion, $2^{\log n - 1}$ operations are performed.

We have $\log n$ layers of recursion.

Question 2

Summing Everything Up

$1 + 2 + 4 + \ldots + 2^{\log n - 1} \leq 2n$ (geometric series)

$= O(n)$

Common Traps

Recursive functions that terminate after a constant number of layers.

```
int func(int n) {
    if (n >= 1000) {
        System.out.println("CS2040");
        return;
    }
    int[] a = new int[n];
    func(n+1);
}
```

```
int func(int n) {
    if (n >= 1) {
        return 0;
    }
    System.out.println("CS2040");
    func(n/2);
    func(n/2);
}
```

Asymptotic Analysis

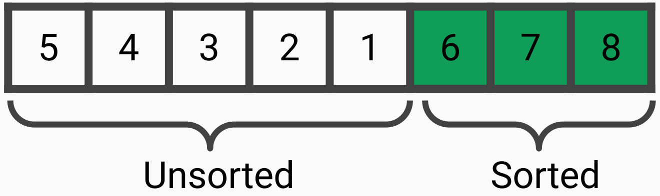
Common Traps

Loops that do not actually do anything or run for constant number of iterations.

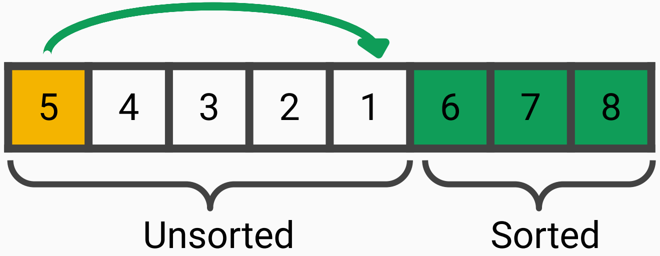
```
public void what_is(int n)  {     // n is large
    if (n > 1) {
        System.out.println("first call: ");
        what_is (n/2);
        for (int i = 0; i < n; i++) {}
            for (int j = n; j > 0; j--)
                System.out.println(n*n + " is n^2");
        System.out.println("second call: ");
        what_is(n/2);
    }
}
```

```
for (int i = 0; i < n; i++) // loop 1
    for (int j = i+1; j > i; j--) // loop 2
        for (int k = n; k > j; k--) // loop 3
            System.out.println("*");
```

Bubble Sort



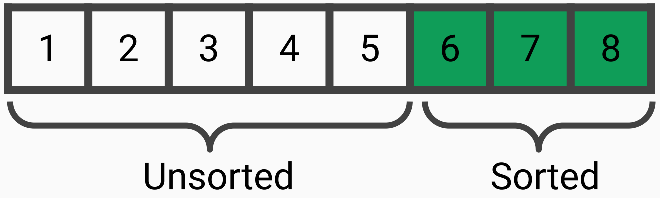
Maintains two regions:
sorted and **unsorted**
Elements in sorted region are in
correct final sorted positions



Each iteration of bubble sort:
biggest element in unsorted region is swapped along the array to correct final sorted position

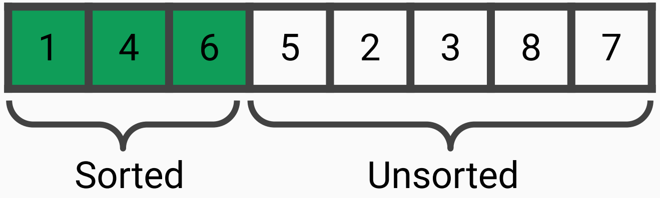
Running Time: O(n^2)

has_swaps = false

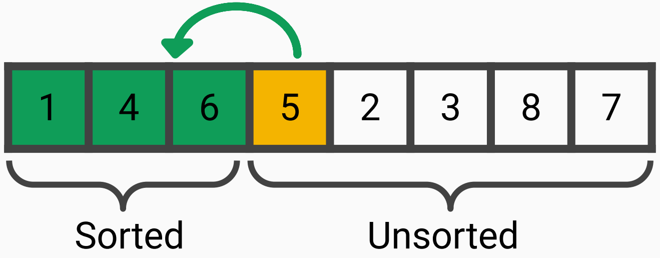


If no swaps made during an iteration, can terminate bubble sort early
Known as **bubble sort with early termination**

Insertion Sort

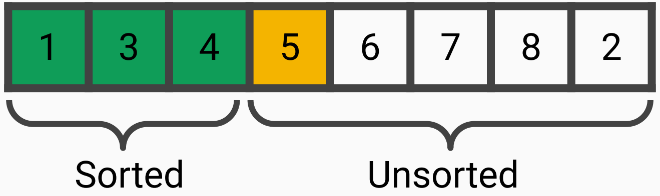


Maintains two regions:
sorted and **unsorted**
Elements in sorted region are **may not be in correct final sorted positions**



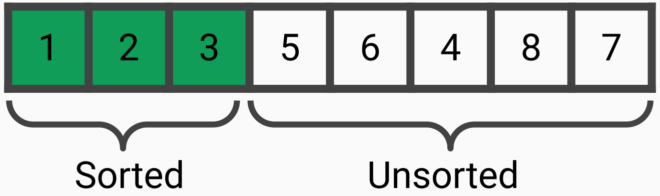
Each iteration of insertion sort:
first element in unsorted region is swapped along the array and inserted in correct position in sorted region

Running Time: O(n^2)
On (nearly) sorted array: O(n)

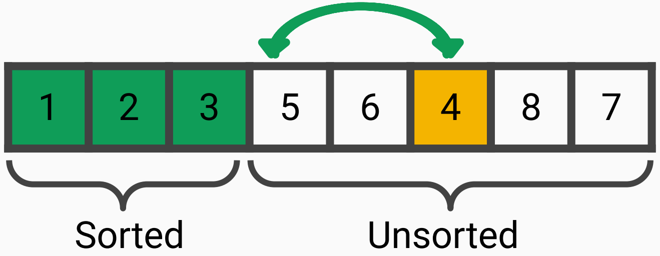


Good for **almost sorted arrays**
If first element in unsorted region is already bigger than all elements in sorted region, **no insertion is needed**

Selection Sort



Maintains two regions:
sorted and **unsorted**
Elements in sorted region are in
correct final sorted positions



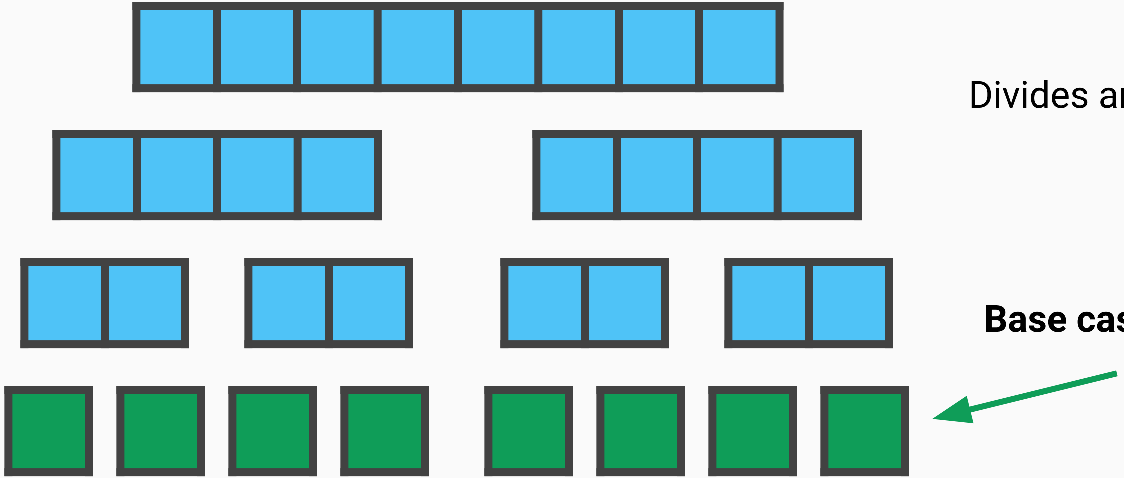
Each iteration of selection sort:
smallest element in unsorted region is swapped with first element in unsorted region to its **correct final sorted position**

Running Time: O(n^2)

Regardless of the initial order of the elements in the array: **will always perform the same number of operations**
Must scan through entire unsorted region to find minimum

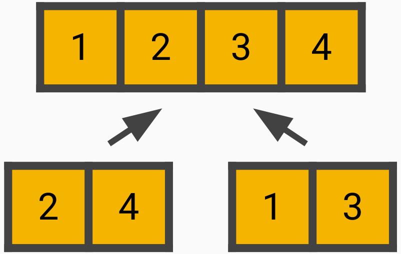
Minimizes swaps, therefore:
Good algorithm if **swaps are expensive**
1 swap / element compared to other algorithms

Merge Sort



Divides arrays **in half recursively**

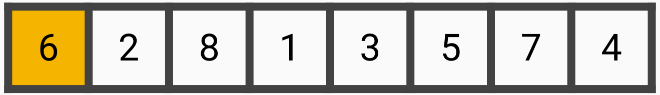
Base case: Arrays of size 1 are already sorted



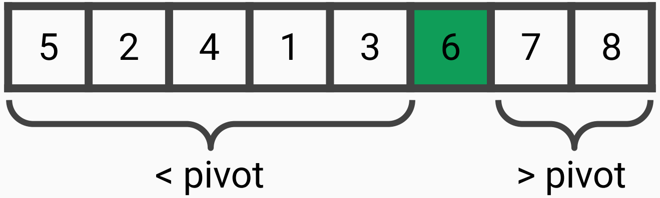
Merge subroutine: Given two sorted arrays, merge them into one sorted array in O(n) time

Running Time: O(n log n)

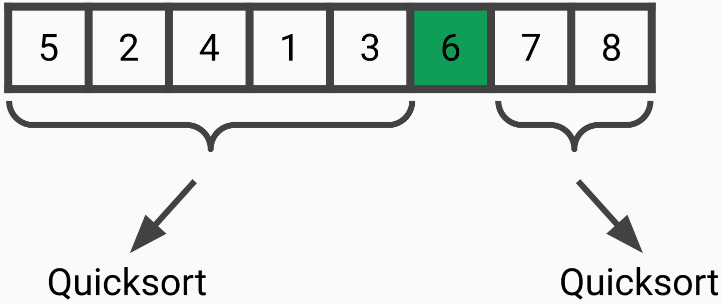
Quicksort



Pick pivot



Partition elements using pivot
Pivot is in **correct final sorted position**



Quicksort **recursively**

Running Time: Expected O(n log n)
Worst Case: O(n^2) for bad pivots

Sorting

In-place

No additional data structures are used to perform the sorting, other than a small number of additional variables.

Stable

Relative order of equal elements is preserved after the sorting is performed.

Lecture 2, Page 53

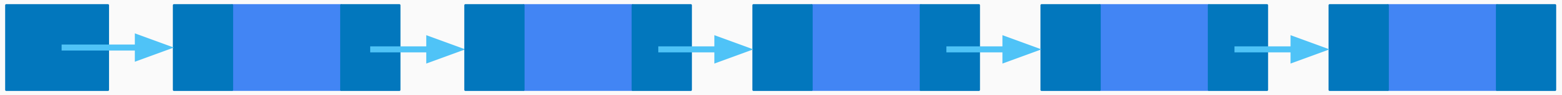
7 Summary of Sorting Algorithms

	Worst Case	Best Case	In-place?	Stable?
Selection Sort	$O(n^2)$	$O(n^2)$	Yes	No
Insertion Sort	$O(n^2)$	$O(n)$	Yes	Yes
Bubble Sort	$O(n^2)$	$O(n^2)$	Yes	Yes
Bubble Sort 2 (improved with flag)	$O(n^2)$	$O(n)$	Yes	Yes
Merge Sort	$O(n \log n)$	$O(n \log n)$	No	Yes
Radix Sort (non-comparison based)	$O(n)$ (see notes 1)	$O(n)$	No	Yes
Quick Sort	$O(n^2)$	$O(n \log n)$	Yes	No

Notes: 1. $O(n)$ for Radix Sort is due to non-comparison based sorting.
2. $O(n \log n)$ is the best possible for comparison based sorting.

Linked List

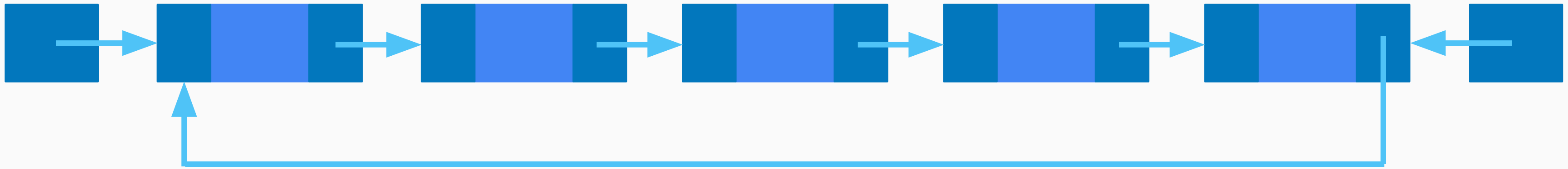
Basic Linked List



Tailed Linked List

Linked List

Circular Linked List



Doubly Linked List

Linked List

Unless the question imposes a restriction on a specific type of linked list that you must use, if you need linked lists, **always use doubly linked lists + tail pointer by default**. Most flexible linked list of them all.

Doubly Linked List

ADT Operations

Stack

push(x): O(1)
pop(): O(1)
peek(): O(1)

Queue

enqueue(x): O(1)
deqeueue(): O(1)
peek(): O(1)

Generally, you are required to make use of the data structures + ADT operations to solve problems. Stack and Queue problems *usually* do not require you to modify the data structures.

Stacks and Queues

Identify **First in, First Out** or **First in, Last Out** properties in problems. Hints towards a stack/queue solution. Make sure you identify FIFO/FILO **correctly**, and not the wrong way round!

Problem exhibits FILO → Use Stack!

Problem exhibits FIFO → Use Queue!

Stack

Last-In, First-Out

Stack exhibits the last-in, first-out property: the last element that is added to the stack is the first one to be removed from the stack.

Any problem exhibiting a last-in, first-out property may possibly be solved using stacks!

Stack

Last-In, First-Out: Bracket Matching

Given a bracket string consisting of ( ) { } [ ], check if all brackets in the string are properly matched.

Observation

$\underbrace{\text{Last bracket to be opened}}_{\text{Last-In}}$ is the $\underbrace{\text{first one to be closed}}_{\text{First-Out}}$. Use a stack!

Stack

Last-In, First-Out: Evaluating Postfix Arithmetic Expressions

Given a postfix arithmetic expression such as `3 6 5 + 4 2 *`, evaluate the expression.

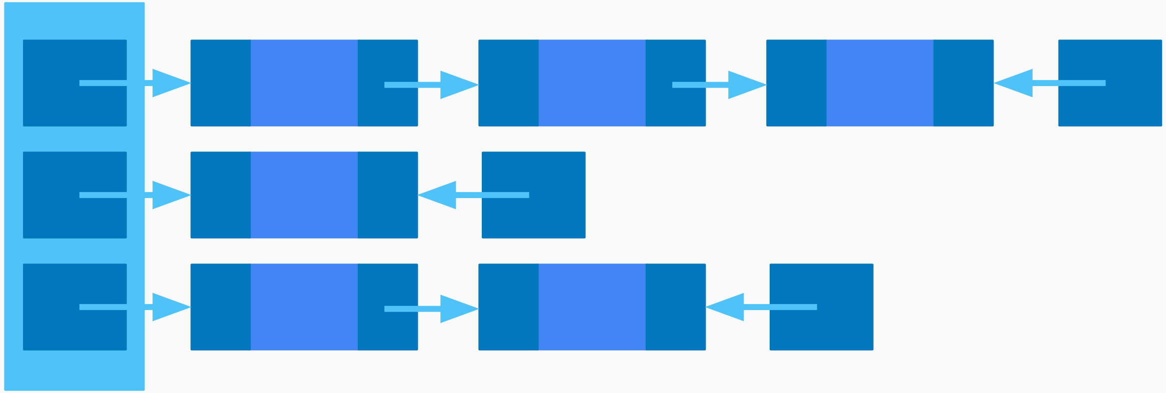
Observation

$\underbrace{\text{Last two operands encountered}}_{\text{Last-In}}$ are the $\underbrace{\text{first to be evaluated.}}_{\text{First-Out}}$ Use a stack!

Collision Resolution

Separate Chaining

Every slot contains a linked list of elements.
When there is a collision, simply add the element to the linked list of the slot the element hashes to.



Collision Resolution

Linear Probing. **Problem: Primary Clustering**

Try the next +1, +2, ..., +*i* slots in succession until an empty slot is found.

Quadratic Probing. **Problem: Secondary Clustering**

Try the next $+1^2$, $+2^2$, ..., $+i^2$ slots in succession until an empty slot is found.
If $\alpha < 0.5$ (table less than half full) and table size is a prime, then quadratic probing will always find empty slot.

Double Hashing

Try the next +h(*key*), +2h(*key*), ..., +*i*h(*key*) slots in succession until an empty slot is found.

Collision Resolution

Properties of a Good Collision Resolution Method

1. Minimize clustering
   - Elements should be ‘spaced out’ after hashing

2. Always finds empty slot
   - No element is rejected by the hash table

3. Fast
   - Finds an empty slot quickly

Hash Functions

Properties of a Good Hash Function

1. Consistent
   - Same key maps to same buckets

Use these to argue whether a given hash function is good.
e.g. Tutorial 4, Question 2

2. Fast to Compute
   - e.g. hash can be computed from key in O(1), O(length of string) for strings

3. Distributes keys as uniformly as possible to buckets
   - Keys are distributed to buckets with equal probability
   - Every bucket has some key hashing to it

Hash Table ADT

Hash Table ADT

Insert(x): O(1)
Delete(x): O(1)
Find/Contains(x): O(1)

Basically, a "black box" that allows you to put stuff in, take stuff out and check if something's there *quickly*.

Very useful if you can reduce parts of a problem to existence checks!

Tutorial 4, Question 4

You are given 4 arrays A, B, C, D, each containing n elements. Check if it is possible to pick one element from each array such that the sum of the 4 elements is 100.

for each element a in A:
    for each element b in B:
        for each element c in C: ⇒ Already know $a + b + c$ at this point
            for each element d in D:
                check if $a + b + c + d = 100$

Question: Is there a d such that $d = 100 - a - b - c$?

Running time: $O(n^4)$

Tutorial 4, Question 4

You are given 4 arrays A, B, C, D, each containing n elements. Check if it is possible to pick one element from each array such that the sum of the 4 elements is 100.

Store all elements in D in a hash table H.

for each element a in A:

    for each element b in B:

        for each element c in C: ⇒ Already know $a + b + c$ at this point

            check if $100 - a - b - c$ is in H

Running time: $O(n^3)$