CSE 250: Midterm Review 3

Lectures 39, 40

Dec 8 and 11, 2023

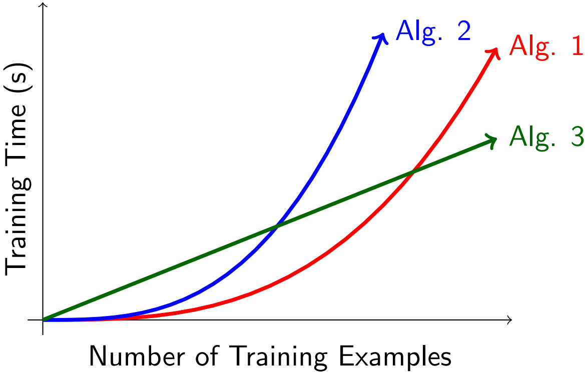
Reminders

- WA3 due Sun, Dec 10
  - PA3 showed you that even 'anonymized' data can be problematic
  - WA3: Look for other cases of problems
- Course Evals Bonus
  - Get to 90% completion across all 3 sections, we'll release an exam question.
  - Section C: 24/112 as of Friday (22%)
- Do you like this class, especially the last 3 lectures?
  - Look at CSE 410 (soon to be CSE 350)
- Do you like/hate this class?
  - Email Eric and me about being a TA!

Exam Day

- **Do** bring...
  - Writing implement (pen or pencil)
  - One note sheet (up to $8\frac{1}{2} \times 11$ inches, double-sided)
- **Do not** bring...
  - Bag (you will be asked to leave it at the front of the room)
  - Computer/Calculator/Watch/etc...
- Wait outside before the exam starts so we can prepare.
  - You will be told when to enter.
- There will be assigned seating.
  - Seating charts will be posted on the doors and projector.
  - See the seat numbers on the chairs.

Runtime



Some Notation

- N: The input "size"
  - How many students I have to email.
  - How many streets on a map.
  - How many key/value pairs in my dictionary
- $T(N)$: The runtime of 'some' implementation of the algorithm.
  - Some... correct implementation.

We care about the "shape" of $T(N)$ when you plot it.

Class Names

$T(N) \in \ldots$

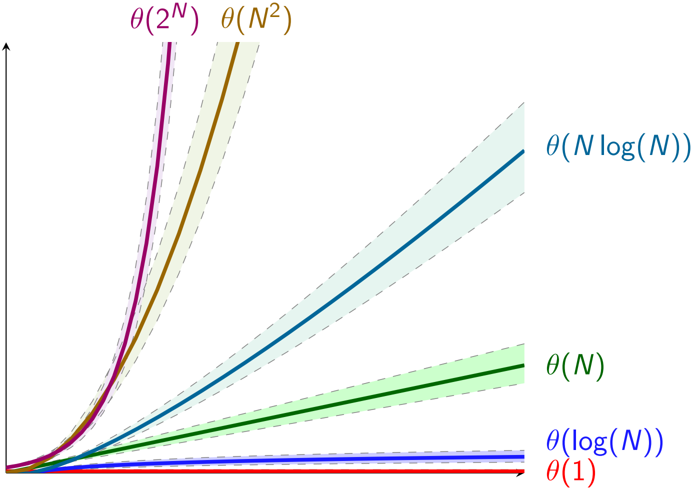
- $\ldots \theta(1)$: Constant
- $\ldots \theta(\log(N))$: Logarithmic
- $\ldots \theta(N)$: Linear
- $\ldots \theta(N \log(N))$: Log-Linear
- $\ldots \theta(N^2)$: Quadratic
- $\ldots \theta(N^k)$ (for any $k \geq 1$): Polynomial
- $\ldots \theta(2^N)$: Exponential

Complexity Bounds

f and g are in the same complexity class if:

- g is bounded from above by something f-shaped
  $g(N) \in O(f(N))$
- g is bounded from below by something f-shaped
  $g(N) \in \Omega(f(N))$

Complexity Classes

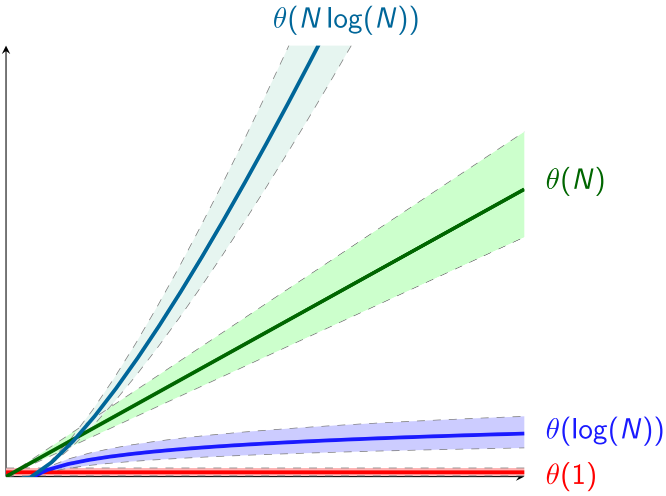


Complexity Bounds

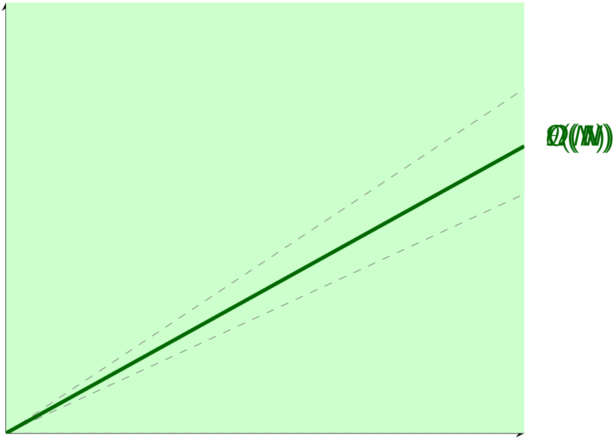
- $O(f(N))$ includes:
  - All functions in $\theta(f(N))$
  - All functions in 'smaller' complexity classes
- $\Omega(f(N))$ includes:
  - All functions in $\theta(f(N))$
  - All functions in 'bigger' complexity classes

$$O(f(N)) \cap \Omega(f(N)) = \theta(f(N))$$

Complexity Bounds



Complexity Bounds



Rules of Thumb

$$O$$

$$\theta$$

$$\Omega$$

Complexity Bounds

$g(N) \in O(f(N))$ (f is an upper bound for g) if and only if:

- You can pick an N_0
- You can pick a c
- For all $N > N_0$: $g(N) \leq c \cdot f(N)$

$g(N) \in \Omega(f(N))$ (f is a lower bound for g) if and only if:

- You can pick an N_0
- You can pick a c
- For all $N > N_0$: $g(N) \geq c \cdot f(N)$

$g(N) \in \theta(f(N))$ if and only if:

- $g(N) \in \Omega(f(N))$
- $g(N) \in O(f(N))$

Rules of Thumb

$$F(N) = f_1(N) + f_2(N) + \ldots + f_k(N)$$

What complexity class is $F(N)$ in?

$f_1(N) + f_2(N)$ is in the greater of $\theta(f_1(N))$ and $\theta(f_2(N))$.

$F(N)$ is in the greatest of any $\theta(f_i(N))$

We say the biggest f_i is the <u>dominant</u> term.

Multi-Class Functions

$$T(N) = \begin{cases} \theta(1) & \textbf{if } N \text{ is even} \\ \theta(N) & \textbf{if } N \text{ is odd} \end{cases}$$

What is the complexity class of $T(N)$?

- $T(N) \in O(N)$ is a **tight** bound.
- $T(N) \in \Omega(1)$ is a **tight** bound.

If the tight Big-O and Big-Ω bounds are different, the function is not in ANY complexity class. (Big-Theta doesn't exist).

Does Big-Theta Exist?

$N + 2N^2$ belongs to one complexity class. $(\theta(N^2))$

$5N + 10N^2 + 2^N$ belongs to one complexity class $(\theta(2^N))$

$\begin{cases} 2^N & \textbf{if } \texttt{rand()} > 0.5 \\ N & \textbf{otherwise} \end{cases}$ does **not** belong to one complexity class.

- **Usually $\theta(f_1(N) + f_2(N) + \ldots)$ is based on the dominant term**
- **If you see cases (i.e., '$\{$'), it's probably multi-class.**

Multi-Class Functions

If...

- $g(N) \in O(f(N))$ is a **tight** upper bound
- $g(N) \in \Omega(f'(N))$ is a **tight** lower bound
- $f'(N) \not\in \theta(f(N))$

... then there is no θ bound for $g(N)$ (g is multi class)

Remember: Addition does <u>not</u> make a function multi-class.

(A tight $\Omega(f(N))$ is the dominant (biggest) term being summed)

Rules of Thumb

- **Lines of Code**: Add Complexities
- **Loops**: Multiply Complexity by the Loop Count
- **If/Then**: Cases block '{'

Bubblesort on Lists

```
public void bubblesort(List[Integer] data)
{
  int N = data.size();
  for(int i = N - 2; i >= 0; i--)
  {
    for(int j = i; j <= N - 1; j++)
    {
      if(data.get(j+1) < data.get(j))
      {
        int temp = data.get(j);
        data.set(j, data.get(j+1));
        data.set(j+1, temp);
      }
    }
  }
}
```

Bubblesort on Lists

```
public void bubblesort(List[Integer] data)
{
  O(N^3)
}
```

Bubblesort on Lists

```
public void bubblesort(List[Integer] data)
{
  int[] array = data.toArray()
  bubblesort(array) // Use the array implementation
  data.clear()
  data.addAll(Arrays.toList(array))
}
```

Bubblesort on Lists

```
public void bubblesort(List[Integer] data)
{
  O(N)
  O(N^2)
  O(N)
  O(N)
}
```

Abstract Data Types

Abstract Data Type defines...

- Domain: What kind of data is stored? (e.g., elements, key/value pairs)
- Constraints: How are items related? (e.g., ordered keys)
- Operations: How can the data be accessed/modified (e.g., 'i'th item)

Like a Java `interface`[1]

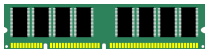
[1] The term `interface` is not quite the same as ADT; The interface only formalizes the permitted operations.

The Sequence ADT

```java
public interface Sequence<E>
{
  public E get(int idx);
  public void set(int idx, E value);
  public int size();
  public Iterator<E> iterator();
}
```

`E` is the type of thing in the Sequence.

CSE 220 Crossover



01001000011001010110110001101100011011111...

01001000 01100101 01101100 01101100 01101111

'H'	'e'	'l'	'l'	'o'

Fixed number of elements

'H'	'e'	'l'	'l'	'o'

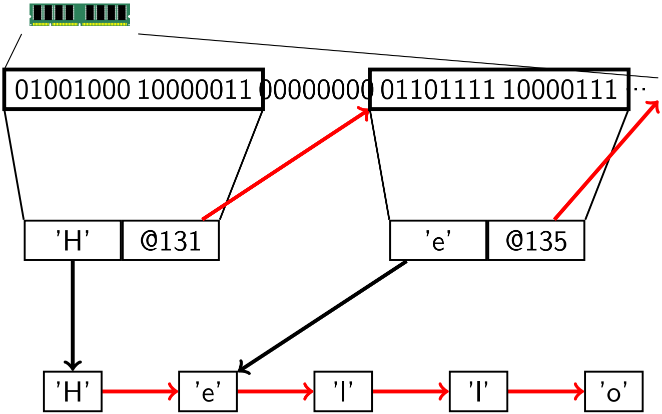
Fixed element size

OpenClipArt: https://freesvg.org/random-access-computer-memory-ram-vector-image

Array

- `public E get(int idx)`
  - Return bytes $\texttt{bPE} \times idx$ to $\texttt{bPE} \times (idx + 1) - 1$
  - $\theta(1)$ (if we treat bPE as a constant)
- `public void set(int idx, E value)`
  - Update bytes $\texttt{bPE} \times idx$ to $\texttt{bPE} \times (idx + 1) - 1$
  - $\theta(1)$ (if we treat bPE as a constant)
- `public int size()`
  - Return `size`
  - $\theta(1)$

CSE 220 Crossover 2: List Harder



OpenClipArt: https://freesvg.org/random-access-computer-memory-ram-vector-image

LinkedList

- `public E get(int idx)`
  - Start at `head`, and move to the `next` element `idx` times. Return the element's `value`.
  - $\theta(idx)$, $O(N)$
- `public void set(int idx, E value)`
  - Start at `head`, and move to the `next` element `idx` times. Update the element's `value`.
  - $\theta(idx)$, $O(N)$
- `public int size()`
  - Start at `head`, and move to the `next` element until you reach the end. Return the number of steps taken.
  - $\theta(N)$

Linked Lists' `size`

Can we do better?

Store size

```
public class LinkedList<T> implements List<T>
{
  LinkedListNode<T> head = null;
  int size = 0;
  /* ... */
}
```

- How expensive is `public int size()` now? ($\theta(1)$)
- How expensive is it to maintain `size`? (Extra $\theta(1)$ work on insert/remove).

Storing redundant information can reduce complexity.

Enumeration

```java
public int sumUpList(LinkedList<Integer> list)
{
  int total = 0;
  int N = list.size()
  Optional<LinkedListNode<Integer>> node = list.head;
  while(node.isPresent())
  {
    int value = node.get().value;
    total += value;
    node = node.get().next;
  }
  return total;
}
```

Enumeration

This code is specialized for `LinkedLists`

- We can't re-use it for an `ArrayList`.
- If we change `LinkedList`, the code breaks.

How do we get code that is both fast and general?

- We need a way to represent a reference to the `idx`'th element of a list.

ListIterator

```
public interface ListIterator<E>
{
  public boolean hasNext();
  public E next();
  public boolean hasPrevious();
  public E previous();
  public void add(E value);
  public void set(E value);
  public void remove();
}
```

Linked Lists

Access list element by index: $O(N)$

Access list element by reference (iterator): $O(1)$

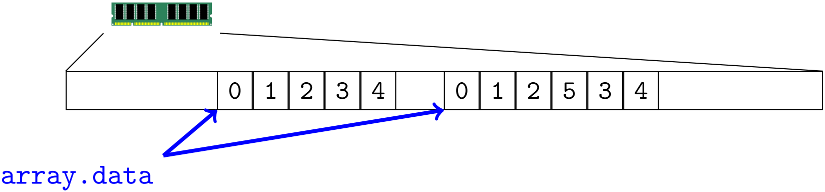
The List ADT

```java
public interface List<E>
  extends Sequence<E> // Everything a sequence has, and...
{
  /** Extend the sequence with a new element at the end */
  public void add(E value);

  /** Extend the sequence by inserting a new element */
  public void add(int idx, E value);

  /** Remove the element at a given index */
  public void remove(int idx);
}
```

Array `add(idx, value)`



`array.add(idx= 2, value= 5)` $\leftarrow \theta(N)$

Idea 1

Idea: Allocate more memory than we need.

ArrayList

Start with a capacity of 2.

1. $\theta(1)$ (size now 1)
2. $\theta(1)$ (size now 2)
3. $2 \cdot \theta(1)$ (capacity now 4; size now 3)
4. $\theta(1)$ (size now 4)
5. $4 \cdot \theta(1)$ (capacity now 8; size now 5)
6. $\theta(1)$ (size now 6)
7. $\theta(1)$ (size now 7)
8. $\theta(1)$ (size now 8)
9. $8 \cdot \theta(1)$ (capacity now 16; size now 9)

…8 more operations before next $\theta(N)$

…16 more operations before next $\theta(N)$

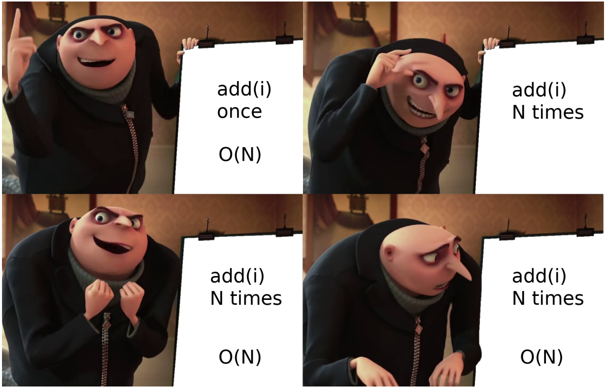
ArrayList

- 2 insertions at $\theta(1)$
- $2 \cdot \theta(1)$ plus 2 insertions at $\theta(1)$ (up to capacity of 4)
- $4 \cdot \theta(1)$ plus 4 insertions at $\theta(1)$ (up to capacity of 8)
- $8 \cdot \theta(1)$ plus 8 insertions at $\theta(1)$ (up to capacity of 16)
- $16 \cdot \theta(1)$ plus 16 insertions at $\theta(1)$ (up to capacity of 32)
- $32 \cdot \theta(1)$ plus 32 insertions at $\theta(1)$ (up to capacity of 64)
- ...

What's the pattern?
($2^i \cdot \theta(1)$ copy on the 2^i'th insertion)

For N insertions, how many copies do we perform?
($\log_2(N)$)

Huh?



Despicable Me; ©2010 Universal Pictures

Amortized Runtimes

$$T_{add}(N) = \begin{cases} \theta(1) & \textbf{if } \textit{capacity} > \textit{size} \\ \theta(N) & \textbf{otherwise} \end{cases}$$

$T_{add}(N) \in O(N)$

- Any **one** call could be $O(N)$
- But the $O(N)$ case happens rarely.
  - ... rarely enough (with doubling) that the expensive call <u>amortizes</u> over the cheap calls.

LinkedList vs ArrayList

```
for(i = 0; i < N; i++)
{
  list.add(i);
}
```

	LinkedList	ArrayList
add(i) once	$O(1)$	$O(N)$
add(i) N times	$O(N)$	$O(N)$

ArrayList.add(i) **behaves like it's** $O(1)$, **but only when it's in a loop.**

Amortized Runtime

- The tight unqualified upper bound on `add(i)` is O(N)
  Any one call to `add(i)` could take up to $O(N)$.
- The tight amortized upper bound on `add(i)` is O(1)
  N calls to `add(i)` average out to $O(1)$ each.
  ($O(N)$ for all N calls)

Amortized Runtime

If $T(N)$ runs in amortized $O(f(N))$, then:

$$\sum_{i=0}^{N} T(N) = \quad N \cdot O(f(N)) = \quad O(N \cdot f(N))$$

Even if $T(N) \not\in O(f(N))$

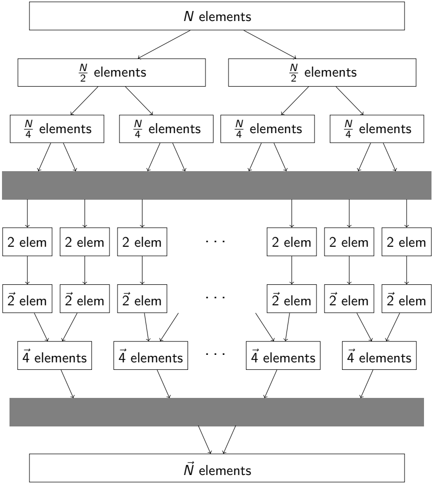
Amortized Runtime

- **Unqualified Bounds**: Always true (no qualifiers)
- **Amortized Bounds**: Only valid in $\sum_{i=0}^{N} T(i)$
  - One call may be expensive, many calls average out cheap

List Runtimes

Op	Array	ArrayList	Linked List (by idx)	Linked List (by iter)
`get(i)`	$\theta(1)$	$\theta(1)$	$\theta(i)$, $O(N)$	$\theta(1)$
`set(i,v)`	$\theta(1)$	$\theta(1)$	$\theta(i)$, $O(N)$	$\theta(1)$
`add(v)`	$\theta(N)$	Amm. $\theta(1)$	$\theta(1)$	$\theta(1)$
`add(i,v)`	$\theta(N)$	$\theta(N)$	$\theta(i)$, $O(N)$	$\theta(1)$
`remove(i)`	$\theta(N)$	$\theta(N)$	$\theta(i)$, $O(N)$	$\theta(1)$

Merge Sort


N elements
$\frac{N}{2}$ elements
$\frac{N}{2}$ elements
$\frac{N}{4}$ elements
$\frac{N}{4}$ elements
$\frac{N}{4}$ elements
$\frac{N}{4}$ elements
2 elem
2 elem
2 elem
...
2 elem
2 elem
2 elem
$\vec{2}$ elem
$\vec{2}$ elem
$\vec{2}$ elem
...
$\vec{2}$ elem
$\vec{2}$ elem
$\vec{2}$ elem
$\vec{4}$ elements
$\vec{4}$ elements
...
$\vec{4}$ elements
$\vec{4}$ elements
$\vec{N}$ elements

Sorting Algorithms

Algorithm	**Runtime**
BubbleSort	$O(N^2)$
MergeSort	Unqualified $O(N \log N)$
QuickSort	Expected $O(N \log N)$
HeapSort	Unqualified $O(N \log N)$

Bound Guarantees

- $f(N)$ is a [Unqualified] Worst-Case Bound $\quad (T(N) \in O(f(N)))$
  The algorithm **always** runs in at most $c \cdot f(N)$ steps.

- $f(N)$ is an Amortized Worst-Case Bound
  N **invocations** of the algorithm **always** run in at most $N \cdot c \cdot f(N)$ steps.

- $f(N)$ is an Expected Worst-Case Bound $\quad (E[T(N)] \in O(f(N)))$
  The algorithm is **statistically likely** to run in at most $c \cdot f(N)$ steps.

Back to Sequence ADTs

- **Sequence**
  - `get(i)`, `set(i, v)`
- **List**
  - … and `add(v)`, `add(i, v)`, `remove(i)`,
- **Stack**
  - `push(v)`, `pop()`, `peek()`
- **Queue**
  - `add(v)`, `remove()`, `peek()`

The Stack ADT

A stack of objects on top of one another.

- **Push**
  Put a new object on top of the stack.
- **Pop**
  Remove the object from the top of the stack.
- **Top**
  Peek at what's on top of the stack.

The Queue ADT

Outside of the US, "queueing" is lining up.

- **Enqueue** (`add(item)` or `offer(item)`)
  Put a new object at the end of the queue.
- **Dequeue** (`remove()` or `poll()`)
  Remove the object from the front of the queue.
- **Peek** (`element()` or `peek()`)
  Peek at what's at the front of the queue.

Queues vs Stacks

- **Queue**
  First in, First out (FIFO)
- **Stack**
  Last in, First out (LIFO, FILO)

Queues vs Stacks (Implementation)

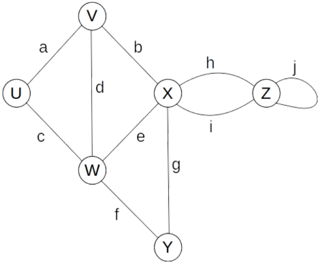
ADT	**Stack**		**Queue**	
using...	`Doub. L. List`	`Array`	`Doub. L. List`	`Array`
`add`	$O(1)$	Amortized $O(1)$	$O(1)$	Amortized $O(1)$
`remove`	$O(1)$	$O(1)$	$O(1)$	$O(1)$

Graphs

A **graph** is a pair (V, E), where

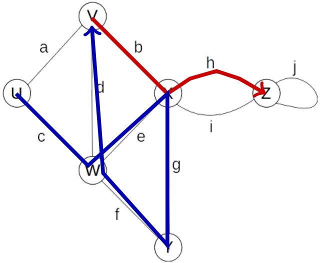
- V is a set of **vertices** (sometimes nodes)
- E is a set of vertex pairs called **edges**
- Edges and vertices may also store data (**labels**)

Graph Terminology



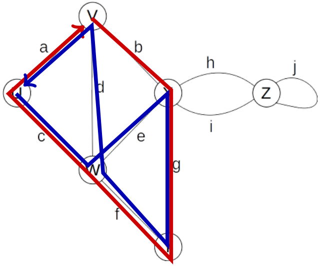
- **Endpoints of an edge**
  U, V are the endpoints of a.
- **Edges incident on a vertex**
  a, b, d are incident on V.
- **Adjacent Vertices**
  U, V are adjacent.
- **Degree of a vertex (# of incident edges)**
  X has degree 5.
- **Parallel Edges** (same endpoints)
  h, i are parallel.
- **Self-loop** (same vertex is start and end)
  j is a self-loop.
- **Simple Graph**
  A graph with no parallel edges or self-loops.

Paths



- **Path**
  A sequence of alternating vertices and edges
  - Begins with a vertex
  - Ends with a vertex
  - Each edge is preceded/followed by its endpoints
- **Simple Path**
  A path that never crosses the same vertex/edge twice
- **Examples**
  V, b, X, h, Z is a simple path.
  $U, c, W, e, X, g, Y, f, W, d, V$ is a path that is not simple.

Cycles



- **Cycle**
  A path that starts and ends on the same vertex.
  - Must contain at least one edge
- **Simple Cycle**
  A cycle where all of the edges and vertices are distinct (except the start/end vertex).
- **Examples**
  $V, b, X, g, Y, f, W, c, U, a, V$ is a simple cycle.
  $U, c, W, e, X, g, Y, f, W, d, V, a, U$ is a cycle that is not simple.

Notation

- N: The number of vertices
- M: The number of edges
- $\texttt{deg}(v)$: The degree of a vertex

Handshake Theorem

$$\sum_{v \in V} \mathtt{deg}(v) = 2M$$

Proof (sketch): Each edge adds 1 to the degree of 2 vertices.

Edge Limit

In a directed graph with no self-loops and no parallel edges:

$$M \leq N \cdot (N - 1)$$

Proof (sketch):

- Each pair is connected at most once (no parallel edges)
- N possible start vertices
- $(N - 1)$ possible end vertices (no self-loops)
- $N \cdot (N - 1)$ distinct combinations possible

The Directed Graph ADT

Interfaces

- `Graph<V, E>`
  - `V`: The vertex label type.
  - `E`: The edge label type.
- `Vertex<V, E>`
  - ... represents a single element (like a LinkedListNode)
  - ... stores a single value of type `V`
- `Edge<V, E>`
  - ... represents an edge (a pair of vertices)
  - ... stores a single value of type `E`

Graph Data Structures

What do we need to store for a graph ((V, E))?

- A collection of vertices
- A collection of edges

Edge List

```
class EdgeList<V, E> implements Graph<V, E>
{
  List<Vertex> vertices = new ArrayList<Vertex>();
  List<Edge>   edges    = new ArrayList<Edge>();

  /*...*/
}
```

Edge List Summary

- `addEdge`, `addVertex`: $O(1)$
- `removeEdge`: $O(1)$
- `removeVertex`: $O(M)$
- `incidentEdges`: $O(M)$
- `hasEdgeTo`: $O(M)$

Space Used: $O(N + M)$
(constant space per vertex, edge)

Improving on the Edge List

How can we avoid searching every edge in the edge list to find the incident edges?

Idea: Store each edges in/out edge list.

Adjacency List

```java
public class Vertex<V, E>
{
  Node<Vertex> node = null;
  List<Edge> inEdges = new BetterLinkedList<Edge>();
  List<Edge> outEdges = new BetterLinkedList<Edge>();
  /*...*/
}
```

Adjacency List Summary

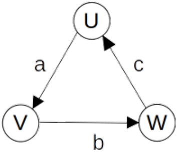
- `addEdge`, `addVertex`: $O(1)$
- `removeEdge`: $O(1)$
- `removeVertex`: $O(\texttt{deg}(v))$
- `incidentEdges`: $O(1) + O(1)$ per `next()`
- `hasEdgeTo`: $O(\texttt{deg}(v))$

Space Used: $O(N + M)$
(constant space per vertex, edge)

The Adjacency Matrix Data Structure

Destination

Origin	U	V	W
U	-	a	-
V	-	-	b
W	c	-	-



Adjacency Matrix Summary

- `addEdge`, `removeEdge`: $O(1)$
- `addVertex`, `removeVertex`: $O(N^2)$
- `incidentEdges`: $O(N)$
- `hasEdgeTo`: $O(1)$

Space Used: $O(N^2)$

A few more definitions

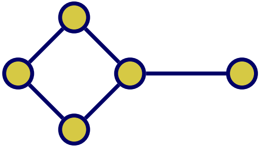
A graph is **connected** if...

- ...there is a path between every pair of vertices.

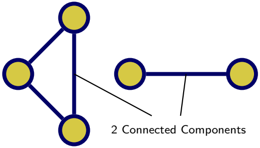
A **connected component** of G is a maximal, connected subgraph of G

- "maximal" means that adding any other vertices from G would break the connected property.
- Any subset of G's edges that makes the subgraph connected is fine.

Connected Graph



Disconnected Graph



Depth First Search (DFS)

Primary Goals

- Visit every vertex in graph $G = (V, E)$.
- Construct a spanning tree for every connected component.
  - **Side Effect**: Compute connected components.
  - **Side Effect**: Compute a path between all connected vertices.
  - **Side Effect**: Determine if the graph is connected.
  - **Side Effect**: Identify any cycles (if they exist).
- Complete in time $O(N + M)$.

Depth First Search (DFS)

`DFS(G)`

Input

- Graph $G = (V, E)$

Output

- Label every edge as a:
  - Spanning Edge: Part of the spanning tree
  - Back Edge: Part of a cycle

Depth First Search (DFS)

`DFSOne(G, v)`

Input

- Graph $G = (V, E)$
- Start vertex $v \in V$

Output

- A spanning tree, rooted at v, to every node in v's connected component.

Depth First Search (DFS)

DFSOne

1. Initialize Todo **Stack** with start vertex v (no edge)
2. Retrieve next todo vertex (or return if none left).
3. If the vertex is already visited[2], return to step 2.
4. Otherwise, mark this vertex as visited.
5. Mark the edge listed in the todo item as a spanning edge.
6. Add todo items for every unvisited, adjacent vertex (via the edge to the current vertex).
7. Return to step 2.

[2] It won't be for DFS or BFS, but bear with me...

Breadth First Search (BFS)

BFSOne

1. Initialize Todo **Queue** with start vertex v (no edge)
2. Retrieve next todo vertex (or return if none left).
3. If the vertex is already visited[3], return to step 2.
4. Otherwise, mark this vertex as visited.
5. Mark the edge listed in the todo item as a spanning edge.
6. Add todo items for every unvisited, adjacent vertex (via the edge to the current vertex).
7. Return to step 2.

[3] It won't be for DFS or BFS, but bear with me...

Dijkstra's Algorithm

Dijkstra One

1. Initialize Todo **Priority Queue** with start vertex v (no edge)
2. Retrieve next todo vertex (or return if none left).
3. If the vertex is already visited, return to step 2.
4. Otherwise, mark this vertex as visited.
5. Mark the edge listed in the todo item as a spanning edge.
6. Add todo items for every unvisited, adjacent vertex (via the edge to the current vertex).
7. Return to step 2.

Graph Traversal

	DFS	BFS	Dijkstra's Algo
Runtime	$O(N+M)$	$O(N+M)$	$O(N+M\log(M))$[4]
Visit Order	Last Visited	Closest by Edge Count	Closest by Total Edge Weight
Spanning Tree	Long paths	Fewest Vertices to Root	Shortest Edge Weight to Root

[4] With Heap as Priority Queue

New ADT: Priority Queue

`PriorityQueue<E>` (`E` must be `Comparable`)

- `public void add(E e)`: Add `e` to the queue.
- `public E peek()`: Return the least element added.
- `public E remove()`: Remove and return the least element added.

(Partial) Ordering Properties

A partial ordering must be...

- **Reflexive** $x \leq x$
- **Antisymmetric** **if** $x \leq y$ **and** $y \leq x$ **then** $x = y$
- **Transitive** **if** $x \leq y$ **and** $y \leq z$ **then** $x \leq z$

(Total) Ordering Properties

A total ordering must be...

- **Reflexive** $x \leq x$
- **Antisymmetric** **if** $x \leq y$ **and** $y \leq x$ **then** $x = y$
- **Transitive** **if** $x \leq y$ **and** $y \leq z$ **then** $x \leq z$
- **Complete** **either** $x \leq y$ **or** $y \leq x$ **for any** $x, y \in A$

Priority Queues

There are two mentalities...

- **Lazy**: Keep everything a mess.
- **Proactive**: Keep everything organized.
- **Balanced**: Keep everything a little sorted.

Lazy Priority Queue

Base Data Structure: Linked List

- `public void add(T v)` $O(1)$
  Append `v` to the end of the linked list.
- `public T remove()` $O(N)$
  Traverse the list to find the least value and remove it.

Proactive Priority Queue

Base Data Structure: Linked List

- `public void add(T v)` $O(N)$
  Traverse the list to insert `v` in sorted order.
- `public T remove()` $O(1)$
  Remove the head of the list.

Binary Min-Heaps

- **Directed** A directed edge in the tree means $\leq$
- **Binary** (max 2 children, easy to reason about)
- **Complete** (every 'level' except last is full)
  - For consistency, keep all nodes in the last level to the left.

This is a **Min-Heap**

Priority Queues

Operation	Lazy	Proactive	Heap
`add`	$O(1)$	$O(N)$	$O(\log(N))$
`remove`	$O(N)$	$O(1)$	$O(\log(N))$
`peek`	$O(N)$	$O(1)$	$O(1)$

Trees

- **Child**
  An adjacent node connected by an out-edge
- **Leaf**
  A node with no children
- **Depth** of a node
  The number of edges from the root to the node
- **Depth** of a tree
  The maximum depth of any node in the tree
- **Level** of a node
  The depth + 1

Tree Traversals

- Pre-order (top-down)
  - visit **root**, visit **left** subtree, visit **right** subtree
- In-order
  - visit **left** subtree, visit **root**, visit **right** subtree
- Post-order (bottom-up)
  - visit **left** subtree, visit **right** subtree, visit **root**

Binary Search Trees

- **Binary** Tree
  - Each element has (at most) 2 children.
- **Binary Search Tree** Constraint
  - Each node has a value.
  - Each node's value is greater than its left descendants
  - Each node's value is lesser than (or equal to) its right descendants
- **Set** Constraint [optional]
  - Each node's value is unique.

Binary Search Trees

Operation	Runtime
`find`	$O(d)$
`insert`	$O(d)$
`remove`	$O(d)$

Balanced Search Trees

- General BST: $d = O(N)$
- **Balanced** BST: $d = O(\log(N))$
  - Complete Tree
  - AVL Tree Property
  - Red-Black Colorability

AVL Trees

- An AVL Tree (<u>A</u>delson-<u>V</u>elsky and <u>L</u>andis) is a BST where every node is "depth balanced"
  - $|\textbf{height}(\mathit{left}) - \textbf{height}(\mathit{right})| \leq 1$

- $\textbf{balance}(v) = \textbf{height}(\mathit{left}) - \textbf{height}(\mathit{right})$

  Maintain $\textbf{balance}(v) \in \{ -1, 0, 1 \}$
  - $\textbf{balance}(b) = 0 \rightarrow$ "v is balanced"
  - $\textbf{balance}(b) = -1 \rightarrow$ "v is left-heavy"
  - $\textbf{balance}(b) = 1 \rightarrow$ "v is right-heavy"

- $\textbf{balance}(v) \in \{ -1, 0, 1 \}$ is the AVL tree property

AVL Trees

If $\textbf{balance}(v) = \textbf{height}(\mathit{left}) - \textbf{height}(\mathit{right})$

Then $N > \textbf{minNodes}(d) = \Omega(1.5^d)$

So $d \in O(\log(N))$

AVL Trees

If the tree starts off balanced:

- The tree can be re-balanced after an insertion in $\log(N)$ time.
- The tree can be re-balanced after a removal in $\log(N)$ time.

Red-Black Trees

A BST is Red-Black Colorable if...

- Every node can be assigned a color, either **Red** or **Black**.
- The root is **Black**.
- The parent of every **Red** node is **Black**.
- The number of **Black** nodes on every path from a null-leaf to the root is the same (the **Black**-depth).

Red-Black Trees

If a BST is red-black colorable...

Then the distance from the root to the shallowest null-leaf is at least half the distance from the root to the deepest null-leaf.

Then The upper "half" of the tree is complete.

Then $N > \textbf{minNodes}(d) = \Omega(2^d)$ **and** $d \in O(\log(N))$

BST Overview

	General BST	**AVL Tree**	**R-B Tree**
`find`	$O(N)$	$O(\log(N))$	$O(\log(N))$
`insert`	$O(N)$	$O(\log(N))$	$O(\log(N))$
`remove`	$O(N)$	$O(\log(N))$	$O(\log(N))$

Note 1: R-B Trees are like AVL Trees, but with a better constant.

How do we implement a set?

- ~~List (Array or Linked)?~~
- ~~Sorted ArrayList?~~
- Balanced Binary Search Tree (AVL, Red-Black) $O(\log N)$
- Hash Tables

Hash Functions

Example Hash Functions

- **SHA256** (used by GIT)
- **MD5**, **BCrypt** (used by unix login, apt)
- **MurmurHash3** (used by Scala)

`hash(e)` is pseudorandom

1. `hash(e)` $\sim$ uniform random value in $[0,$ `Integer.MAX_VALUE`$)$
2. `hash(e)` always returns the same value for the same `e`
3. `hash(e)` is uncorrelated with `hash(e')` for `e` $\neq$ `e'`

Hash Functions

`hash(e)` is ...

- Pseudorandom ("Evenly distributed" over $[0, B)$)
- Deterministic (Same value every time)

HashSet

- `public boolean add(E a)`
  Insert the element into the list at $\texttt{hash}(a) \mod B$.
  Expected $O\left(\frac{N}{B}\right)$
- `public boolean remove(E a)`
  Find the element in the list at $\texttt{hash}(a) \mod B$ and remove it.
  Expected $O\left(\frac{N}{B}\right)$
- `public boolean contains(E a)`
  Find the element in the list at $\texttt{hash}(a) \mod B$.
  Expected $O\left(\frac{N}{B}\right)$
- `public int size()`
  Return a pre-computed size. $O(1)$

Expected Bucket Size

After N insertions, how many records can we <u>expect</u> in the average bucket?

Let X_j be the number of records in bucket j.

After N insertions $0 \leq X_j \leq N$:

- $X_j = 0$ with $P[X_j = 0] = ???$
- $X_j = 1$ with $P[X_j = 1] = ???$
- $X_j = 2$ with $P[X_j = 2] = ???$
- $\ldots$
- $X_j = N$ with $P[X_j = N] = ???$

Expected Bucket Size

For N insertions, we repeat the process: $X_{0,j}, X_{1,j}, X_{2,j}, \ldots X_{N,j}$

$$\mathbb{E}[\sum_i X_{i,j}] = \mathbb{E}[X_{0,j}] + \mathbb{E}[X_{1,j}] + \ldots + \mathbb{E}[X_{N,j}]$$
$$= \underbrace{\frac{1}{B} + \ldots + \frac{1}{B}}_{N \text{ times}}$$
$$= \frac{N}{B}$$

- **Expected** Runtime of `insert`, `find`, `remove`: $O\left(\frac{N}{B}\right)$
- **Unqualified** Runtime of `insert`, `find`, `remove`: $O(N)$

Resizing the Hash Table

- Rehashes required: $\leq \log(N)$.
- The ith rehashing $O(2^i)$ work.
- Total work after N insertions is no more than...

$$\sum_{i=0}^{\log(N)} O(2^i) = O\left(\sum_{i=0}^{\log(N)} 2^i\right)$$
$$= O\left((2^{\log(N)+1} - 1)\right)$$
$$= O\left(N\right)$$

- Work per insertion (amortized): $O\left(\frac{N}{N}\right) = O(1)$
  (plus the cost of actually inserting into the linked list)

Resizing the Hash Table (Rehashing)

Remember the load factor $\alpha = \frac{N}{B}$

The expected runtime of `insert`, `find`, `remove` is $O(\alpha)$

If we can ensure that $\alpha \leq \alpha_{max}$ for some constant α_{max}, then $O(\alpha) = O(1)$

After enough inserts to make $\alpha > \alpha_{max}$ (with B buckets):

- Create a new hash table with $2B$ buckets.
- Insert every element e from the original table into the new one according to $\texttt{hash}(e) \mod 2B$

Recap: get(x)

Expected Cost

- Find the bucket $O(c_{hash})$[5]
- Find the record in the bucket $O(\alpha \cdot c_{equals})$[6]

Total: $O(c_{hash} + \alpha c_{equals}) = O(1 + 1) = O(1)$

Unqualified Worst-Case Cost

- Find the bucket $O(c_{hash})$
- Find the record in the bucket $O(N \cdot c_{equals})$

Total: $O(c_{hash} + N \cdot c_{equals}) = O(1 + N) = O(N)$

[5] c_{hash} is the cost of the hash function.
[6] c_{equals} is the cost of `.equals`.

Recap: add(x)

Expected Cost

- Find the bucket $O(c_{hash})$
- Find the record in the bucket $O(\alpha \cdot c_{equals})$
- Replace the existing record or append it to the list $O(1)$

Total: $O(c_{hash} + \alpha c_{equals} + 1) = O(1 + 1 + 1) = O(1)$

Unqualified Worst-Case Cost

- Find the bucket $O(c_{hash})$
- Find the record in the bucket $O(N \cdot c_{equals})$
- Replace the existing record or append it to the list $O(1)$

Total: $O(c_{hash} + N \cdot c_{equals} + 1) = O(1 + N + 1) = O(N)$

Recap: remove(x)

Expected Cost

- Find the bucket $O(c_{hash})$
- Find the record in the bucket $O(\alpha \cdot c_{equals})$
- Remove the record from the linked list $O(1)$

Total: $O(c_{hash} + \alpha c_{equals} + 1) = O(1 + 1 + 1) = O(1)$

Unqualified Worst-Case Cost

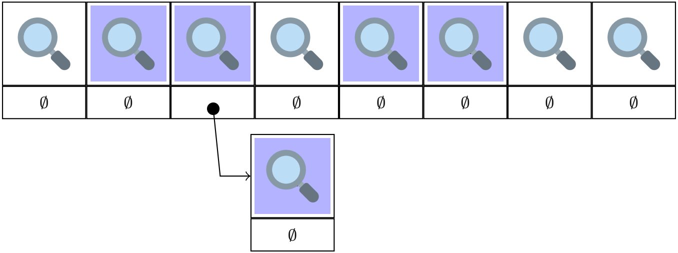
- Find the bucket $O(c_{hash})$
- Find the record in the bucket $O(N \cdot c_{equals})$
- Remove the record from the linked list $O(1)$

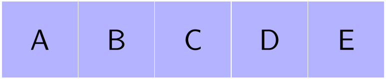
Total: $O(c_{hash} + N \cdot c_{equals} + 1) = O(1 + N + 1) = O(N)$

Hash Table Operations

Operation	Unqualified	Amortized	Expected
`add`	$O(N)$	$O(N)$	$O(1)$
`remove`	$O(N)$	$O(N)$	$O(1)$
`contains/get`	$O(N)$	$O(N)$	$O(1)$

Iterating over a Hash Table





Iterating over a Hash Table

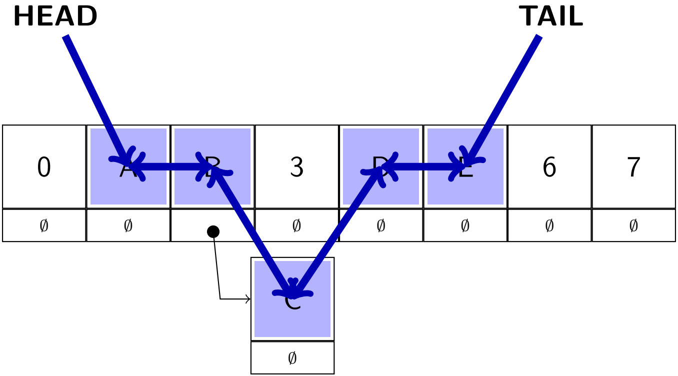
- Visit every hash bucket $O(B)$
- Visit every element in every hash bucket $O(N)$

Total: $O(B + N)$

Linked Hash Table

Idea: Organize the hash table elements in a linked list

Linked Hash Table



Iterating over a Linked Hash Table

- Visit every element via linked list $O(N)$

Total: $O(N)$ (no more $O(B)$ factor)

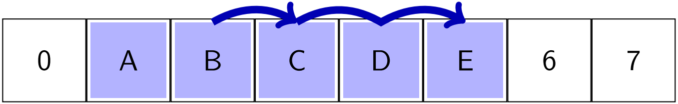
<u>Insert</u> (Changes only)

- Append the new element to the tail of the linked list. $O(1)$

<u>Remove</u> (Changes only)

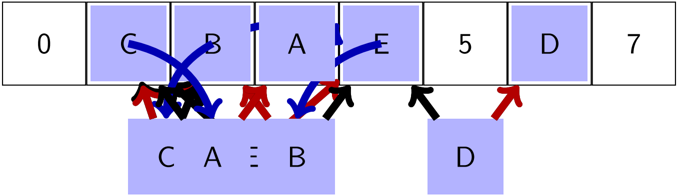
- Remove the element from its position in the linked list. $O(1)$

Hash Table with Open Addressing




```
hash(A) = 1
hash(B) = 2
hash(C) = 2
hash(D) = 4
hash(E) = 3
```

Cuckoo Hashing



$\texttt{hash}_1\texttt{(A) = 1; hash}_2\texttt{(A) = 3}$

$\texttt{hash}_1\texttt{(B) = 2; hash}_2\texttt{(B) = 4}$

$\texttt{hash}_1\texttt{(C) = 2; hash}_2\texttt{(C) = 1}$

$\texttt{hash}_1\texttt{(D) = 4; hash}_2\texttt{(D) = 6}$

$\texttt{hash}_1\texttt{(E) = 1; hash}_2\texttt{(E) = 4}$

Cuckoo Hashing

Find $O(1)$

- Look at array index $\texttt{hash}_1 \mod B$
- Look at array index $\texttt{hash}_2 \mod B$

Cuckoo Hashing

- **Find** is unqualified $O(1)$
- **Remove** is unqualified $O(1)$
- **Insert** is expected $O(1)$ (for low values of α)

Dynamic Hashing

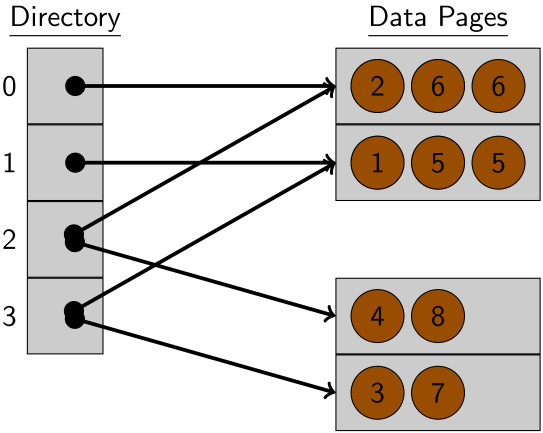
Observation: If $a = N \mod B$ then either

- $a = N \mod 2B$, or
- $a + B = N \mod 2B$

Doubling the size of the hash table always rehashes every element in a specific bucket to one of two places.

Dynamic Hashing

Example: $\texttt{hash}(N) = N$



Dynamic Hashing

- An array (of size B) of pointers to arrays (each of size α). (and some book-keeping metadata)
- When doubling the array size, only copy the array pointers. (faster than rehashing the entire hash table)
- Only split one bucket at a time
- Only double the array when a bucket being split has only one pointer to it.

A Dynamic Hash Table does not have better asymptotic complexity than a Hash Table with Chaining (but has a better constant factor).