CSE 250
Data Structures

Dr. Eric Mikida
epmikida@buffalo.edu
208 Capen Hall

Course Recap/Final Review

Announcements

- WA5 due tomorrow at midnight
- Final Exam on Monday May 13th – See Piazza for more info
- Fill out your course evaluations!
- Comments about the TAs? Fill out a TA evaluation as well!

Course Roadmap

Analysis Tools/Techniques	ADTs	Data Structures
Asymptotic Analysis, (Unqualified) Runtime Bounds		
	Sequence	Array, LinkedList
Amortized Runtime	List	ArrayList, LinkedList
Recursive analysis, divide and conquer, Average/Expected Runtime		
	Stack, Queue	ArrayList, LinkedList
Midterm #1		

Course Roadmap

Analysis Tools/Techniques	ADTs	Data Structures
Review recursive analysis	Graphs, PriorityQueue	EdgeList, AdjacencyList, AdjacencyMatrix
	Trees	BST, AVL Tree, Red-Black Tree, Heaps
Midterm #2		
Review expected runtime	HashTables	
Miscellaneous		

Analysis Tools and Techniques

Recap of Runtime Complexity

Big-Θ — Tight Bound

- Growth functions are in the **same** complexity class
- If f(n) ∈ **Θ**(g(n)) then an algorithm taking f(n) steps is as "exactly" as fast as one that takes g(n) steps.

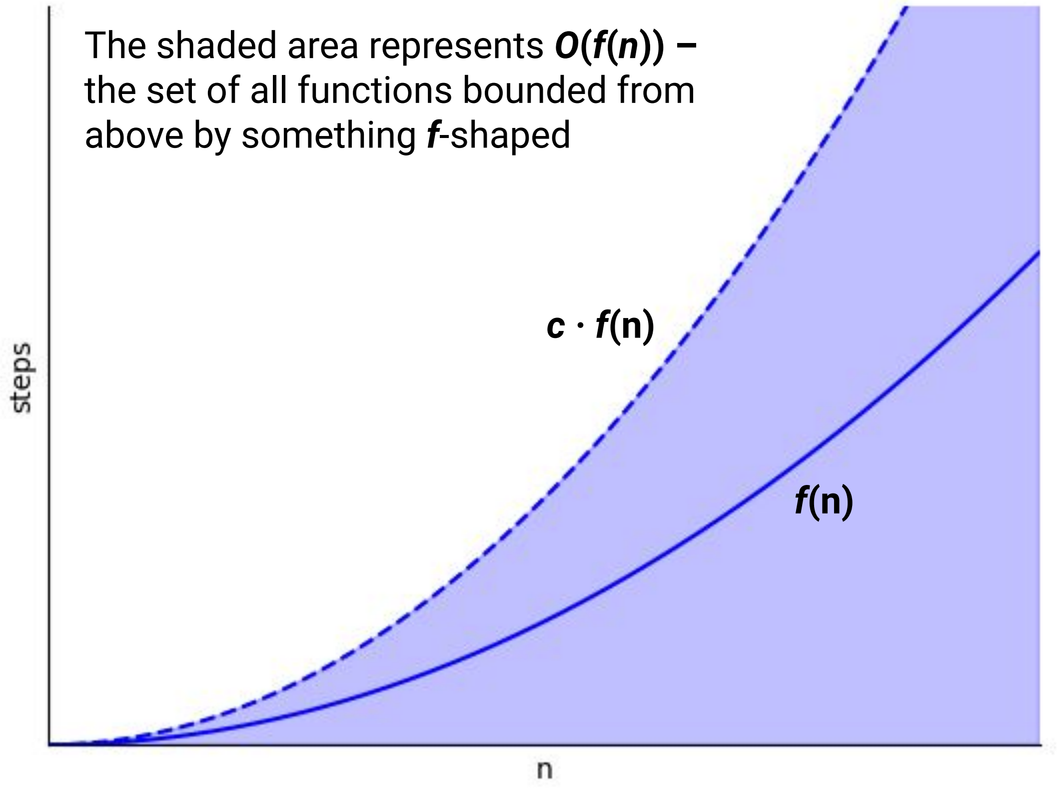
Big-O — Upper Bound

- Growth functions in the **same or smaller** complexity class.
- If f(n) ∈ O(g(n)), then an algorithm that takes f(n) steps is *at least as fast* as one taking g(n) (but it may be even faster).

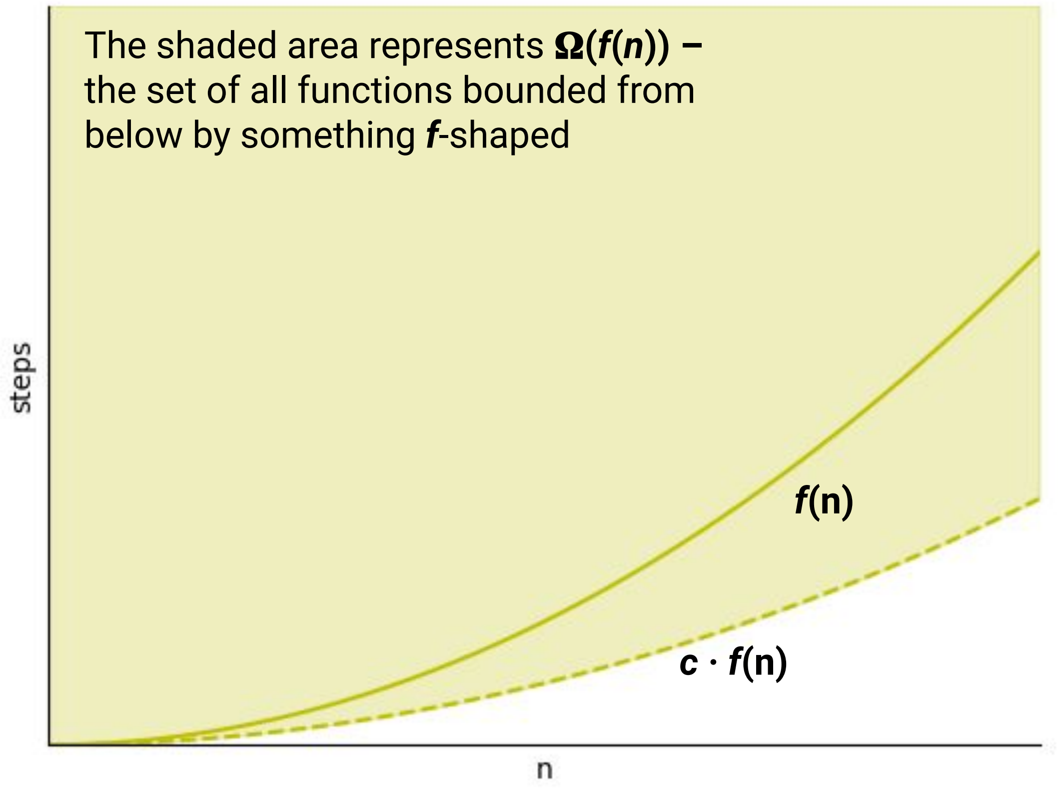
Big-Ω — Lower Bound

- Growth functions in the **same or bigger** complexity class
- If f(n) ∈ **Ω**(g(n)), then an algorithm that takes f(n) steps is *at least as slow* as one that takes g(n) steps (but it may be even slower)

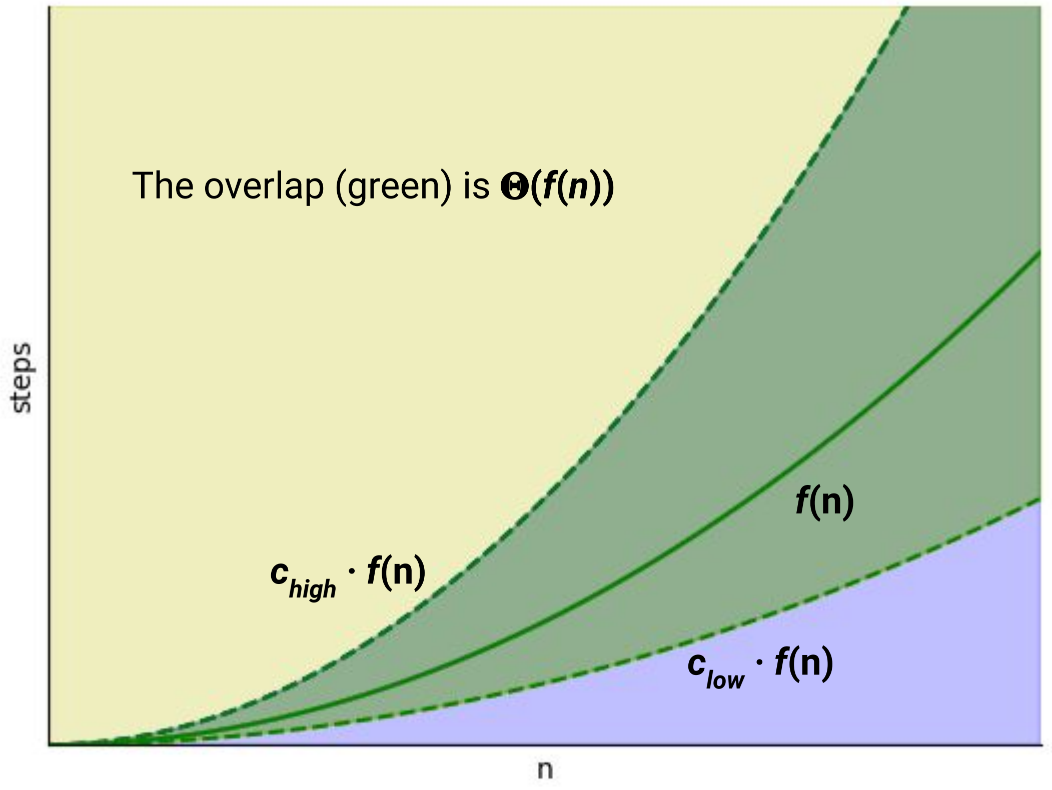
Bounded from Above: Big O



Bounded from Below: Big Ω

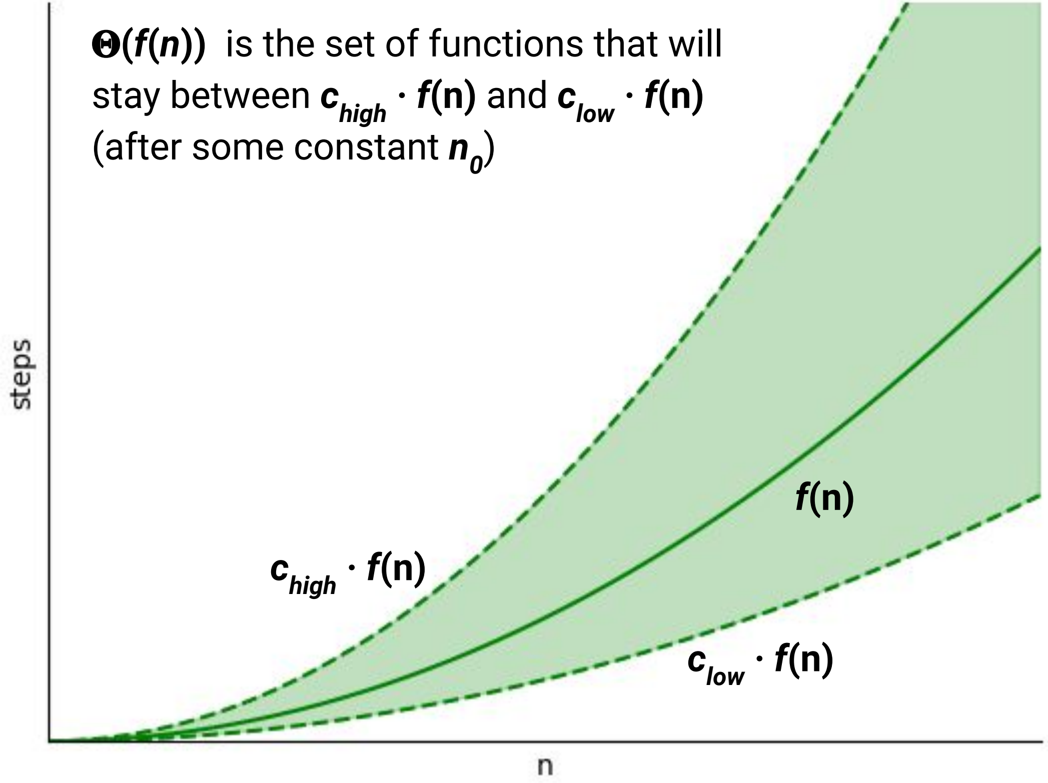


Complexity Class: Big Θ

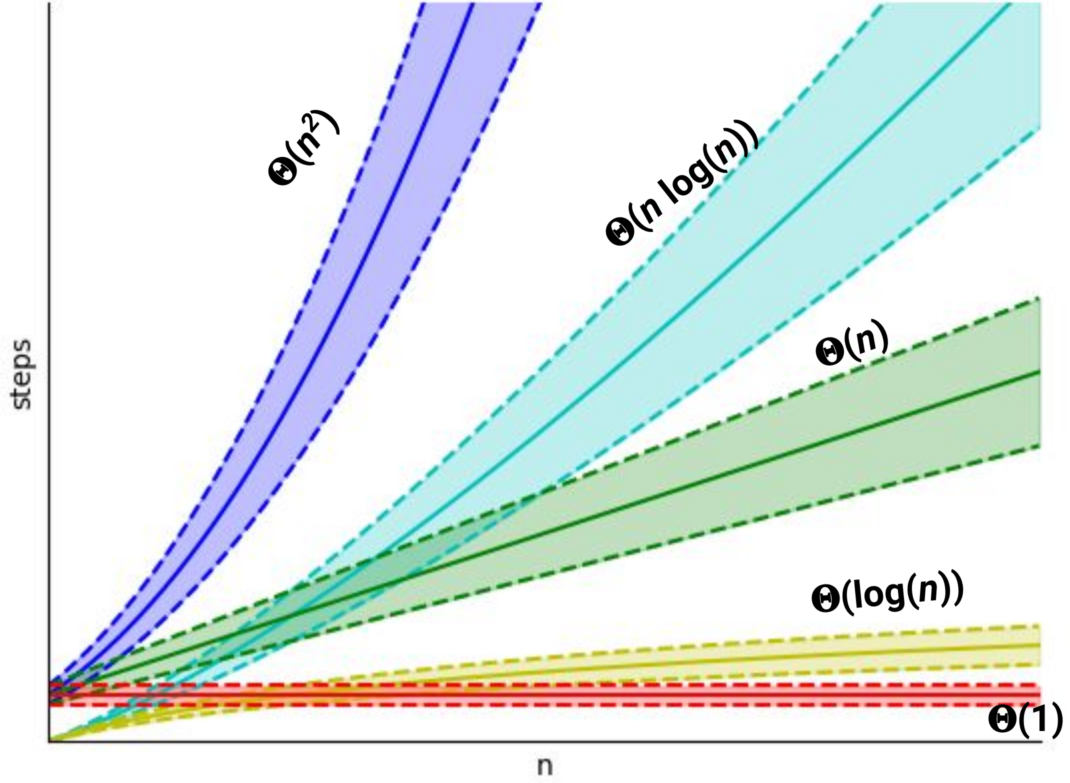


Complexity Class: Big Θ

Θ(***f***(***n***)) is the set of functions that will stay between $\boldsymbol{c_{high}} \cdot \boldsymbol{f}(\mathbf{n})$ and $\boldsymbol{c_{low}} \cdot \boldsymbol{f}(\mathbf{n})$ (after some constant $\boldsymbol{n_0}$)



Complexity Class Ranking



$\Theta(1) < \Theta(\log(n)) < \Theta(n) < \Theta(n \log(n)) < \Theta(n^2) < \Theta(n^3) < \Theta(2^n)$

Common Runtimes (in order of complexity)

Constant Time:	$\Theta(1)$
Logarithmic Time:	$\Theta(\log(n))$
Linear Time:	$\Theta(n)$
Quadratic Time:	$\Theta(n^2)$
Polynomial Time:	$\Theta(n^k)$ for some $k > 0$
Exponential Time:	$\Theta(c^n)$ (for some $c \geq 1$)

Formal Definitions

f(n) ∈ *O*(g(n)) iff exists some constants **c**, $\mathbf{n_0}$ s.t.

f(n) ≤ c * g(n) for all **n >** $\mathbf{n_0}$

f(n) ∈ Ω(g(n)) iff exists some constants **c**, $\mathbf{n_0}$ s.t.

f(n) ≥ c * g(n) for all **n >** $\mathbf{n_0}$

f(n) ∈ Θ(g(n)) iff **f(n) ∈ *O*(g(n))** and **f(n) ∈ Ω(g(n))**

Shortcut

What complexity class do each of the following belong to:

$f(n) = 4n + n^2 \in \Theta(n^2)$

$g(n) = 2^n + 4n \in \Theta(2^n)$

$h(n) = 100\, n \log(n) + 73n \in \Theta(n \log(n))$

Shortcut: Just consider the complexity of the most dominant term

Multi-class Functions

$$T(n) = \begin{cases} n^2 & \text{if } n \text{ is even} \\ n & \text{if } n \text{ is odd} \end{cases}$$

It is not bounded from above by ***n***, therefore it cannot be in **Θ(*n*)**

It is not bounded from below by ***n*²**, therefore it cannot be in **Θ(*n*²)**

What is the tight upper bound of this function? $\boldsymbol{T(n) \in O(n^2)}$

What is the tight lower bound of this function? $\boldsymbol{T(n) \in \Omega(n)}$

What is the complexity class of this function? It does not have one!

Amortized Runtime

If n calls to a function take $\Theta(f(n))$...

We say the **<u>Amortized Runtime</u>** is $\Theta(f(n) / n)$

The **<u>amortized runtime</u>** of `add` on an `ArrayList` is: $\Theta(n/n) = \Theta(1)$
The **<u>unqualified runtime</u>** of `add` on an `ArrayList` is: $O(n)$

Algorithms with Randomness

What about algorithms with a random component, ie QuickSort?

QuickSort: Worst-Case Runtime

What is the worst-case runtime?

$$T_{quicksort}(n) \in O(n^2)$$

Remember: This is called the unqualified runtime…we don't take any extra context into account

QuickSort: Worst-Case Runtime

Is the worst case runtime representative?

No! (the actual runtime will almost always be faster)

But what **can** we say about runtime?

QuickSort Runtime

Now we can write our runtime function in terms of random variables:

$$T(n) = \begin{cases} \Theta(1) & \textbf{if } n \leq 1 \\ T(0) + T(n-1) + \Theta(n) & \textbf{if } n > 1 \land X = 1 \\ T(1) + T(n-2) + \Theta(n) & \textbf{if } n > 1 \land X = 2 \\ T(2) + T(n-3) + \Theta(n) & \textbf{if } n > 1 \land X = 3 \\ .. & \\ T(n-2) + T(1) + \Theta(n) & \textbf{if } n > 1 \land X = n-1 \\ T(n-1) + T(0) + \Theta(n) & \textbf{if } n > 1 \land X = n \end{cases}$$

QuickSort Runtime

...and convert it to the expected runtime over the variable ***X***

$$E[T(n)] = \begin{cases} \Theta(1) & \textbf{if } n \leq 1 \\ E[T(X-1)] + E[T(n-X)] + \Theta(n) & \textbf{otherwise} \end{cases}$$

This looks like the runtime of MergeSort, so now our hypothesis is that our **Expected Runtime** is ***n*** **log(*n*)**

What guarantees do you get?

If $f(n)$ is a Tight Bound (Big-Θ)

The algorithm **always** runs in $\boldsymbol{cf(n)}$ steps

← Unqualified runtime

If $f(n)$ is a Worst-Case Bound (Big-O)

The algorithm **always** runs in *at most* $\boldsymbol{cf(n)}$

If $f(n)$ is an Amortized Bound

$\boldsymbol{n}$ invocations of the algorithm **always** run in $\boldsymbol{cnf(n)}$ steps

If $f(n)$ is an Average/Expected Bound

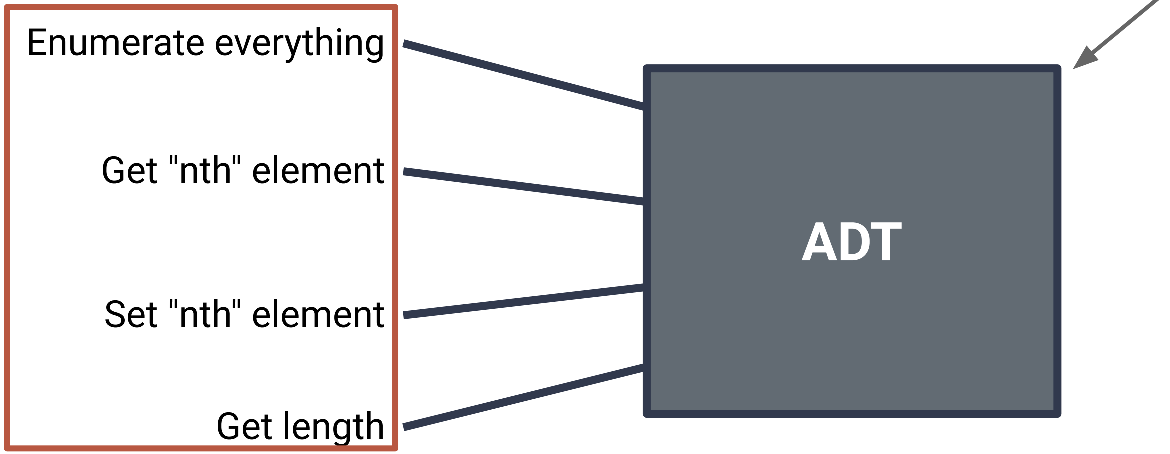
…we don't have any guarantees

ADTs and Data Structures

Abstract Data Types (ADTs)

The specification of **what** a data structure can do

What's in the box? …we don't know, and in some sense…we don't care



Usage is governed by **what** we can do, not **how** it is done

Abstract Data Type vs Data Structure

ADT

The interface to a data structure

*Defines **what** the data structure can do*

Many data structures can implement the same ADT

Data Structure

The implementation of one (or more) ADTs

*Defines **how** the different tasks are carried out*

Different data structures will excel at different tasks

Abstract Data Type vs Data Structure

ADT

The interface to ...

*Defines **what** the ...*

can ...

Many data st...

implement th...

Data Structure

...tation of one (or ...) ADTs

...e different tasks ...ried out

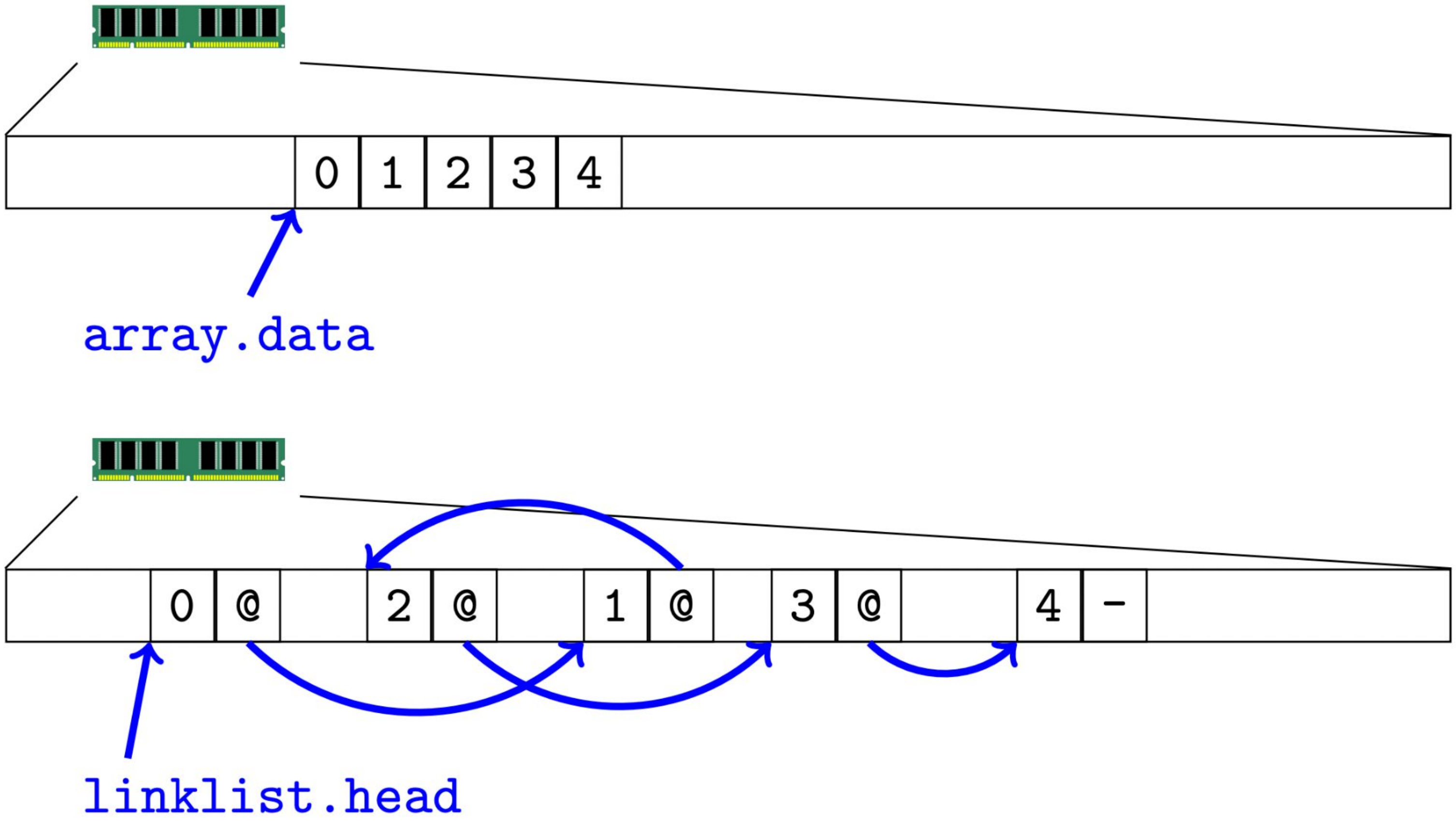
...uctures will excel at different tasks

Think about the Linked List we implemented for PA1.

The internal structure and the mental model of our sequence are very different.

The Sequence ADT

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



Arrays and Linked Lists in Memory

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);
}
```

Runtime Summary

	ArrayList	Linked List (by index)	Linked List (by reference)
`get(...)`	$\Theta(1)$	$\Theta(\text{idx})$ or $O(n)$	$\Theta(1)$
`set(...)`	$\Theta(1)$	$\Theta(\text{idx})$ or $O(n)$	$\Theta(1)$
`size()`	$\Theta(1)$	$\Theta(1)$	$\Theta(1)$
`add(...)`	$O(n)$, Amortized $\Theta(1)$	$\Theta(\text{idx})$ or $O(n)$	$\Theta(1)$
`remove(...)`	$O(n)$	$\Theta(\text{idx})$ or $O(n)$	$\Theta(1)$

Stacks and Queues

Variants on Sequences (more ADTs)

`Stack`
- LIFO: last in first out
- push elements to the top of the stack
- pop elements from the top of the stack

`Queue`
- FIFO: first in first out
- enqueue elements to the end of the queue
- dequeue elements from the front of the queue

`PriorityQueue`
- Elements ordered by ***priority***
- dequeue removes the highest priority element (smallest element in Java)

Recap

Stacks: Last In First Out (LIFO)

- Push (put item on top of the stack) $\Theta(1)$ (or amortized $O(1)$)
- Pop (take item off top of stack) $\Theta(1)$
- Peek (peek at top of stack) $\Theta(1)$

Queues: First in First Out (FIFO)

- Enqueue (put item on the end of the queue) $\Theta(1)$ (or amortized $O(1)$)
- Dequeue (take item off the front of the queue) $\Theta(1)$
- Peek (peek at the item in the front of the queue) $\Theta(1)$

Stacks and Queues can be easily implemented with Arrays and Linked Lists. PriorityQueues can be...but not very efficiently...we'll get back to that when we see Trees

Graphs

A (Directed) Graph ADT

Two type parameters (`Graph[V,E]`)

`V`: The vertex label type

`E`: The edge label type

Vertices

…are elements

…store a value of type `V`

Edges

…are also elements

…store a value of type `E`

A (Directed) Graph ADT

What can we do with a Graph?

- Iterate through the vertices
- Iterate through the edges
- Add a vertex
- Add an edge
- Remove a vertex
- Remove an edge

A (Directed) Graph ADT

```java
public interface Graph<V, E> {
    public Iterator<Vertex> vertices();
    public Iterator<Edge> edges();
    public Vertex addVertex(V label);
    public Edge addEdge(Vertex orig, Vertex dest, E label);
    public void removeVertex(Vertex vertex);
    public void removeEdge(Edge edge);
}
```

A (Directed) Graph ADT

What can we do with a Vertex?

- Get it's label
- Get the outgoing edges
- Get the incoming edges
- Get all incident edges
- Check if it's adjacent to another Vertex

A (Directed) Graph ADT

What can we do with an Edge?

- Get it's label
- Get the incident vertices

A (Directed) Graph ADT

```
public interface Vertex<V,E> {
  public V getLabel();
  public Iterator<Edge> getOutEdges();
  public Iterator<Edge> getInEdges();
  public Iterator<Edge> getIncidentEdges();
  public boolean hasEdgeTo(Vertex v);
}

public interface Edge<V,E> {
  public Vertex getOrigin();
  public Vertex getDestination();
  public E getLabel();
}
```

Implementation Attempt 1: Edge List

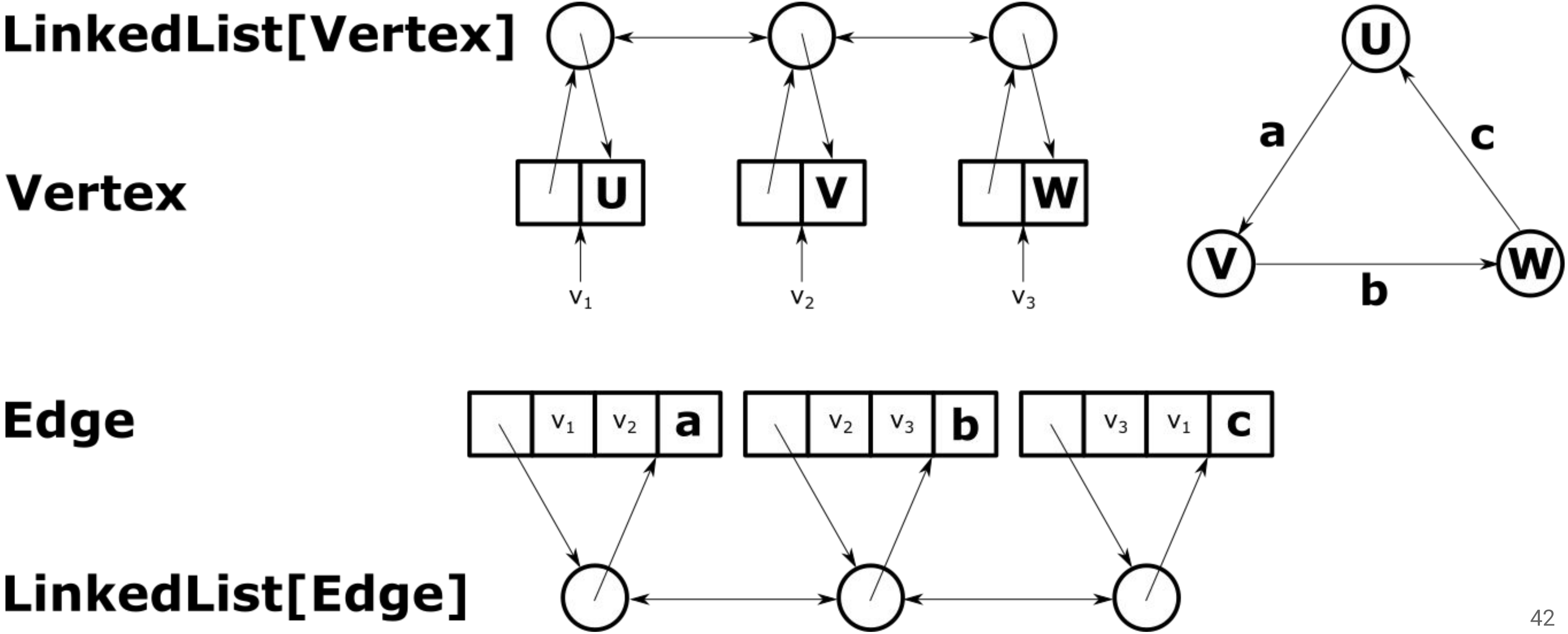
<u>Data Model:</u>

A List of Edges
(LinkedList)

A List of Vertices
(LinkedList)

An EdgeList is exactly what it sounds like, a single big list of edges (with a list of vertices as well)

Edge List Summary



Edge List Summary

- `addEdge`, `addVertex`: $O(1)$
- `removeEdge`: $O(1)$
- `removeVertex`: $O(m)$
- `vertex.incidentEdges`: $O(m)$
- `vertex.edgeTo`: $O(m)$
- **Space Used:** $O(n) + O(m)$

← Involves checking every edge in the graph

How can we improve?

Idea: Store the in/out edges for each vertex!

(Called an adjacency list)

Adjacency List Summary

Graph

vertices: LinkedList[Vertex]
edges: LinkedList[Edge]

Vertex

label: T
node: LinkedListNode
inEdges: LinkedList[Edge]
outEdges: LinkedList[Edge]

Edge

label: T
node: LinkedListNode
inNode: LinkedListNode
outNode: LinkedListNode

Storing the list of incident edges in the vertex saves us the time of checking every edge in the graph.

The edge now stores additional nodes to ensure removal is still $\Theta(1)$

Adjacency List Summary

- `addEdge, addVertex`: $\Theta(1)$
- `removeEdge`: $\Theta(1)$
- `removeVertex`: $\Theta(deg(vertex))$
- `vertex.incidentEdges`: $\Theta(deg(vertex))$
- `vertex.edgeTo`: $\Theta(deg(vertex))$
- **Space Used: $\Theta(n) + \Theta(m)$**

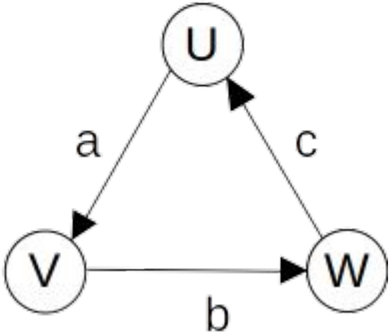
Now we already know what edges are incident without having to check them all

Adjacency Matrix

Destination

Origin

	U	V	W
U	*-*	***a***	*-*
V	*-*	*-*	***b***
W	***c***	*-*	*-*



Adjacency Matrix Summary

- `addEdge, removeEdge`: $\Theta(1)$
- `addVertex, removeVertex`: $\Theta(n^2)$
- `vertex.incidentEdges`: $\Theta(n)$
- `vertex.edgeTo`: $\Theta(1)$
- **Space Used:** $\Theta(n^2)$

Depth-First Search

<u>Primary Goals</u>

- 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 cycles
- Complete in time $O(|V| + |E|)$

DFSOne

```
public void DFSOne(Graph graph, Vertex v) {
  v.setLabel(VISITED);
  for (Edge e : v.outEdges) {
    if (e.label == UNEXPLORED) {
      Vertex w = e.to;
      if (w.label == UNEXPLORED) {
        e.setLabel(SPANNING);
        DFSOne(graph, w);
      } else {
        e.setLabel(BACK);
      }
    }
}}
```

DFSOne

```
public void DFSOne(Graph graph, Vertex v) {
  v.setLabel(VISITED); ← Mark the vertex as VISITED (so we'll never try to visit it again)
  for (Edge e : v.outEdges) {
    if (e.label == UNEXPLORED) {
      Vertex w = e.to;
      if (w.label == UNEXPLORED) {
        e.setLabel(SPANNING);
        DFSOne(graph, w);
      } else {
        e.setLabel(BACK);
      }
    }
}}
```

DFSOne

```
public void DFSOne(Graph graph, Vertex v) {
  v.setLabel(VISITED);
  for (Edge e : v.outEdges) {
    if (e.label == UNEXPLORED) {
      Vertex w = e.to;
      if (w.label == UNEXPLORED) {
        e.setLabel(SPANNING);
        DFSOne(graph, w);
      } else {
        e.setLabel(BACK);
      }
    }
}}
```

Check every outgoing edge (every possible way we could leave the current vertex)

DFSOne

```
public void DFSOne(Graph graph, Vertex v) {
  v.setLabel(VISITED);
  for (Edge e : v.outEdges) {
    if (e.label == UNEXPLORED) {
      Vertex w = e.to;
      if (w.label == UNEXPLORED) {
        e.setLabel(SPANNING);
        DFSOne(graph, w);
      } else {
        e.setLabel(BACK);
      }
    }
}}
```

Follow the unexplored edges

DFSOne

```
public void DFSOne(Graph graph, Vertex v) {
  v.setLabel(VISITED);
  for (Edge e : v.outEdges) {
    if (e.label == UNEXPLORED) {
      Vertex w = e.to;
      if (w.label == UNEXPLORED) {
        e.setLabel(SPANNING);
        DFSOne(graph, w);
      } else {
        e.setLabel(BACK);
      }
    }
}}
```

If it leads to an unexplored vertex, then it is a spanning edge. Recursively explore that vertex.

DFSOne

```
public void DFSOne(Graph graph, Vertex v) {
  v.setLabel(VISITED);
  for (Edge e : v.outEdges) {
    if (e.label == UNEXPLORED) {
      Vertex w = e.to;
      if (w.label == UNEXPLORED) {
        e.setLabel(SPANNING);
        DFSOne(graph, w);
      } else {
        e.setLabel(BACK);
      }
    }
}}
```

Otherwise, we just found a cycle

Depth-First Search Complexity

In summary…

1.	Mark the vertices **`UNVISITED`**	$O(\|V\|)$
2.	Mark the edges **`UNVISITED`**	$O(\|E\|)$
3.	**`DFS`** vertex loop	**$O(\|V\|)$ iterations**
4.	All calls to **`DFSOne`**	**$O(\|E\|)$ total**
		$O(\|V\| + \|E\|)$

We can also implement DFS without recursion by using a Stack!

Breadth-First Search

Primary Goals

- Visit every vertex in graph ***G* = (*V*,*E*)** **in increasing order of distance from the start**
- 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 cycles
  - **Side Effect: Identify shortest paths to the starting vertex**
- Complete in time $O(|V| + |E|)$, with memory overhead $O(|V|)$

```java
public void BFSOne(Graph graph, Vertex v) {
  Queue<Vertex> todo = new Queue<>();
  v.setLabel(VISITED);
  todo.enqueue(v);
  while (!todo.isEmpty()) {
    Vertex curr = todo.dequeue();
    for (Edge e : curr.outEdges) {
      if (e.label == UNEXPLORED) {
        Vertex w = e.to;
        if (w.label == UNEXPLORED) {
          w.setLabel(VISITED);
          e.setLabel(SPANNING);
          todo.enqueue(w);
        } else {
          e.setLabel(CROSS);
        }
}}}}
```

```
public void BFSOne(Graph graph, Vertex v) {
  Queue<Vertex> todo = new Queue<>();
  v.setLabel(VISITED);
  todo.enqueue(v);
  while (!todo.isEmpty()) {
    Vertex curr = todo.dequeue();
    for (Edge e : curr.outEdges) {
      if (e.label == UNEXPLORED) {
        Vertex w = e.to;
        if (w.label == UNEXPLORED) {
          w.setLabel(VISITED);
          e.setLabel(SPANNING);
          todo.enqueue(w);
        } else {
          e.setLabel(CROSS);
        }
}}}}
```

Use a queue to keep track of what vertices we want to visit (basically a running TODO list)

```
public void BFSOne(Graph graph, Vertex v) {
  Queue<Vertex> todo = new Queue<>();
  v.setLabel(VISITED);
  todo.enqueue(v);
  while (!todo.isEmpty()) {
    Vertex curr = todo.dequeue();
    for (Edge e : curr.outEdges) {
      if (e.label == UNEXPLORED) {
        Vertex w = e.to;
        if (w.label == UNEXPLORED) {
          w.setLabel(VISITED);
          e.setLabel(SPANNING);
          todo.enqueue(w);
        } else {
          e.setLabel(CROSS);
        }
}}}}
```

Dequeue a vertex from the Queue and check all of it's outgoing edges

```java
public void BFSOne(Graph graph, Vertex v) {
  Queue<Vertex> todo = new Queue<>();
  v.setLabel(VISITED);
  todo.enqueue(v);
  while (!todo.isEmpty()) {
    Vertex curr = todo.dequeue();
    for (Edge e : curr.outEdges) {
      if (e.label == UNEXPLORED) {
        Vertex w = e.to;
        if (w.label == UNEXPLORED) {
          w.setLabel(VISITED);
          e.setLabel(SPANNING);
          todo.enqueue(w);
        } else {
          e.setLabel(CROSS);
        }
}}}}
```

When we find a new vertex, mark it as VISITED, and add it to our TODO list.

Remember, our TODO list is a Queue (FIFO) so whatever we enqueud first will be the next thing we dequeue (and explore)

```java
public void BFSOne(Graph graph, Vertex v) {
  Queue<Vertex> todo = new Queue<>();
  v.setLabel(VISITED);
  todo.enqueue(v);
  while (!todo.isEmpty()) {
    Vertex curr = todo.dequeue();
    for (Edge e : curr.outEdges) {
      if (e.label == UNEXPLORED) {
        Vertex w = e.to;
        if (w.label == UNEXPLORED) {
          w.setLabel(VISITED);
          e.setLabel(SPANNING);
          todo.enqueue(w);
        } else {
          e.setLabel(CROSS);
        }
}}}}
```

When doing BFS we label edges that return to visited vertices as CROSS edges

Breadth-First Search Complexity

In summary…

1.	Mark the vertices **`UNVISITED`**	$O(\lvert V \rvert)$
2.	Mark the edges **`UNVISITED`**	$O(\lvert E \rvert)$
3.	Add each vertex to the work queue	$O(\lvert V \rvert)$
4.	Process each vertex	$O(\lvert E \rvert)$ **total**

$$O(|V| + |E|)$$

Djikstra's Algorithm

- Both BFS and DFS search the whole graph
  - DFS – Exploration order based on a Stack (LIFO)
  - BDS – Exploration order based on a Queue (FIFO)
  - The paths BFS finds are the shortest paths **in terms of # of edges**
- Djikstra's Algorithm finds the shortest path in terms of total distance
  - Can't rely on Stack or Queue – need an ADT that orders the vertices

```java
public void Djikstras(Graph graph, Vertex v) {
  PriorityQueue<TodoEntry> todo = new PriorityQueue<>();
  todo.add(new TodoEntry(v,0));
  while (!todo.isEmpty()) {
    TodoEntry curr = todo.poll();
    if (curr.vertex.label == UNEXPLORED) {
      curr.vertex.setLabel(VISITED);
      for (Edge e : curr.vertex.outEdges) {
        Vertex w = e.to;
        if (w.label == UNEXPLORED) {
          todo.add(new TodoEntry(w, curr.weight + e.weight));
        }
      }
    }
  }
}
```

Create a new PriorityQueue and insert the starting point with a distance of 0

```java
public void Djikstras(Graph graph, Vertex v) {
  PriorityQueue<TodoEntry> todo = new PriorityQueue<>();
  todo.add(new TodoEntry(v,0));
  while (!todo.isEmpty()) {
    TodoEntry curr = todo.poll();
    if (curr.vertex.label == UNEXPLORED) {
      curr.vertex.setLabel(VISITED);
      for (Edge e : curr.vertex.outEdges) {
        Vertex w = e.to;
        if (w.label == UNEXPLORED) {
          todo.add(new TodoEntry(w, curr.weight + e.weight));
        }
      }
    }
  }
}
```

When we pull something out of the PriorityQueue, if it is still UNEXPLORED then we just found the shortest path to that vertex, and we can mark it as VISITED

```java
public void Djikstras(Graph graph, Vertex v) {
  PriorityQueue<TodoEntry> todo = new PriorityQueue<>();
  todo.add(new TodoEntry(v,0));
  while (!todo.isEmpty()) {
    TodoEntry curr = todo.poll();
    if (curr.vertex.label == UNEXPLORED) {
      curr.vertex.setLabel(VISITED);
      for (Edge e : curr.vertex.outEdges) {
        Vertex w = e.to;
        if (w.label == UNEXPLORED) {
          todo.add(new TodoEntry(w, curr.weight + e.weight));
        }
      }
    }
  }
}
```

Add each unexplored neighbor to the PriorityQueue.
Set it's distance equal to our current distance plus the weight of the edge to get to the neighbor.

```java
public void Djikstras(Graph graph, Vertex v) {
  PriorityQueue<TodoEntry> todo = new PriorityQueue<>();
  todo.add(new TodoEntry(v,0));
  while (!todo.isEmpty()) {
    TodoEntry curr = todo.poll();
    if (curr.vertex.label == UNEXPLORED) {
      curr.vertex.setLabel(VISITED);
      for (Edge e : curr.vertex.outEdges) {
        Vertex w = e.to;
        if (w.label == UNEXPLORED) {
          todo.add(new TodoEntry(w, curr.weight + e.weight));
        }
      }
    }
  }
}
```

What is the complexity?

```java
public void Djikstras(Graph graph, Vertex v) {
  PriorityQueue<TodoEntry> todo = new PriorityQueue<>();
  todo.add(new TodoEntry(v,0));
  while (!todo.isEmpty()) {
    TodoEntry curr = todo.poll();
    if (curr.vertex.label == UNEXPLORED) {
      curr.vertex.setLabel(VISITED);
      for (Edge e : curr.vertex.outEdges) {
        Vertex w = e.to;
        if (w.label == UNEXPLORED) {
          todo.add(new TodoEntry(w, curr.weight + e.weight));
        }
      }
    }
  }
}
```

We know removal from a PriorityQueue is ***O*(log(`todo.size()`))**

How big can **`todo`** get?

What is the complexity?

```
public void Djikstras(Graph graph, Vertex v) {
  PriorityQueue<TodoEntry> todo = new PriorityQueue<>();
  todo.add(new TodoEntry(v,0));
  while (!todo.isEmpty()) {
    TodoEntry curr = todo.poll();
    if (curr.vertex.label == UNEXPLORED) {
      curr.vertex.setLabel(VISITED);
      for (Edge e : curr.vertex.outEdges) {
        Vertex w = e.to;
        if (w.label == UNEXPLORED) {
          todo.add(new TodoEntry(w, curr.weight + e.weight));
        }
      }
    }
  }
}
```

We know removal from a PriorityQueue is ***O*(log(`todo.size()`))**

How big can **`todo`** get? |E|

Each vertex may be added once per incoming edge. So the size of the PriorityQueue can get as large as |E|

```java
public void Djikstras(Graph graph, Vertex v) {
  PriorityQueue<TodoEntry> todo = new PriorityQueue<>();
  todo.add(new TodoEntry(v,0));
  while (!todo.isEmpty()) {
    TodoEntry curr = todo.poll();
    if (curr.vertex.label == UNEXPLORED) {
      curr.vertex.setLabel(VISITED);
      for (Edge e : curr.vertex.outEdges) {
        Vertex w = e.to;
        if (w.label == UNEXPLORED) {
          todo.add(new TodoEntry(w, curr.weight + e.weight));
        }
      }
    }
  }
}
```

We know removal from a PriorityQueue is ***O*(log(`todo.size()`))**

How big can **`todo`** get? |E|

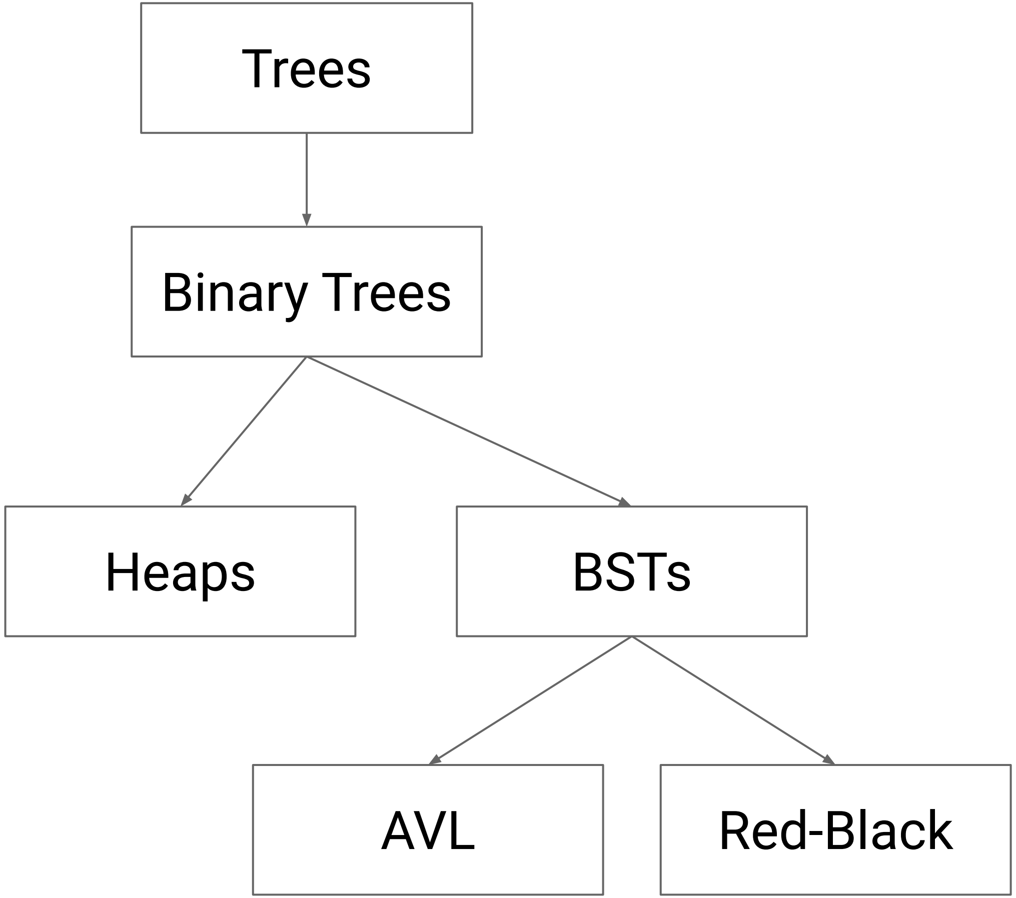
Label the |V| vertices

|E| adds/removes to the PriorityQueue

What is the complexity? O(|V| + |E| log(|E|))

Trees

Types of Trees Covered



Binary Min Heaps

Organize our priority queue as a directed tree

Directed: A directed edge from ***a*** to ***b*** means that $a \leq b$

A max heap would reverse this ordering

Binary: Max out-degree of 2 (easy to reason about)

Complete: Every "level" except the last is full (from left to right)

Balanced: TBD (basically, all leaves are roughly at the same level)

This makes it easy to encode into an array (later today)

The `MinHeap` ADT

`void pushHeap(T value)`
Place an item into the heap

`T popHeap()`
Remove and return the minimal element from the heap

`T peek()`
Peek at the minimal element in the heap

`int size()`
The number of elements in the heap

pushHeap

Idea: Insert the element at the next available spot, then fix the heap.

1. Call the insertion point **`current`**
2. While **`current != root`** and **`current < parent`**
   a. Swap **`current`** with **`parent`**
   b. Set **`current = parent`**

What is the complexity (or how many swaps occur)? **O(log(*n*))**

popHeap

Idea: Replace root with the last element then fix the heap

1. Start with `current = root`
2. While `current` has a `child < current`
    a. Swap `current` with its smallest `child`
    b. Set `current = child`

What is the complexity (or how many swaps occur)? **O(log(*n*))**

Priority Queues

Operation	Lazy	Proactive	Heap
add	$O(1)$	$O(n)$	$O(\log(n))$
poll	$O(n)$	$O(1)$	$O(\log(n))$
peek	$O(n)$	$O(1)$	$O(1)$

Storing heaps

Notice that:

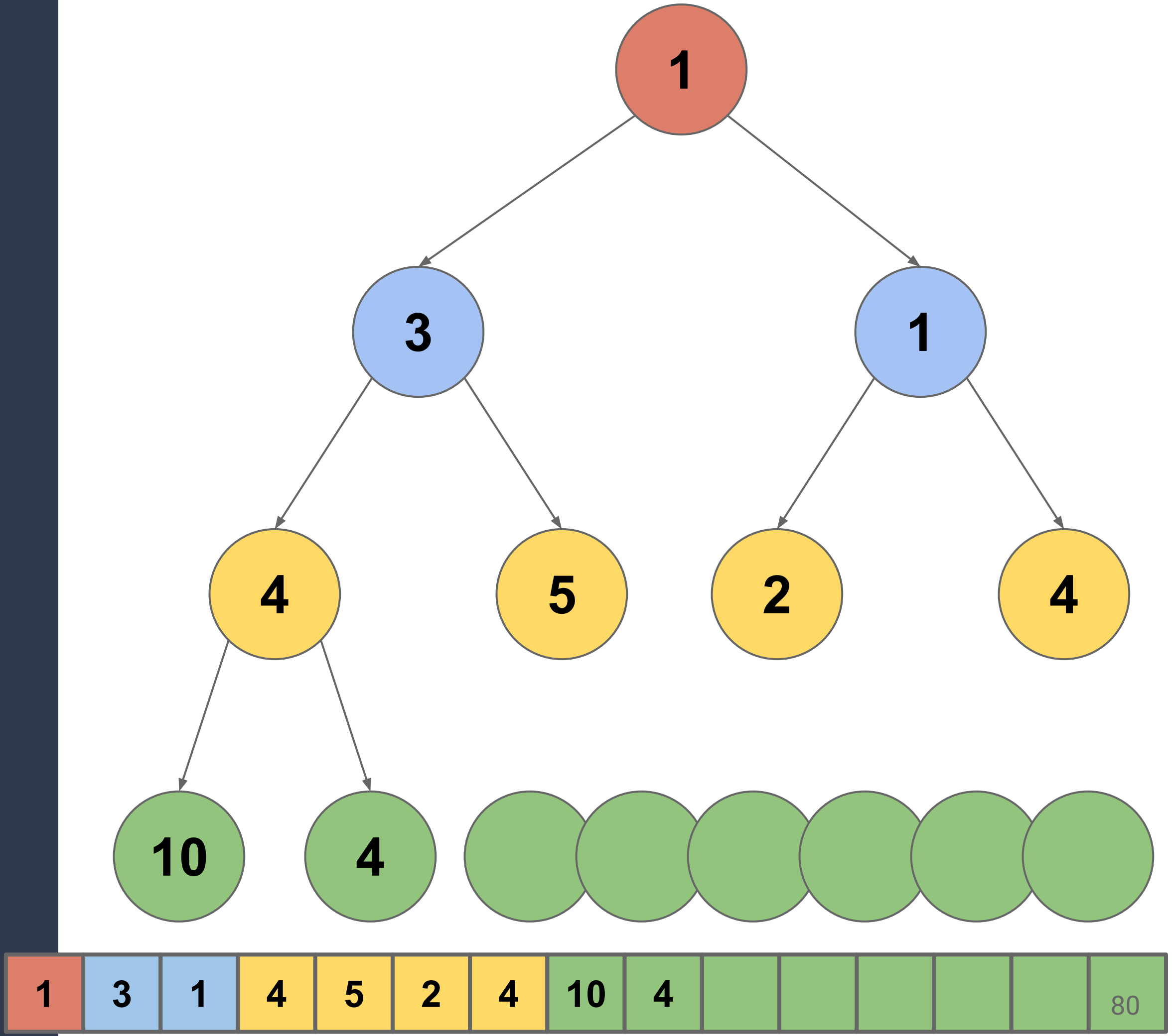
1. Each level has a maximum size
2. Each level grows left-to-right
3. Only the last layer grows

How can we compactly store a heap?

Idea: Use an **`ArrayList`**

Storing Heaps

How can we store this heap in an array buffer?



Binary Search Tree

A **Binary Search Tree** is a **Binary Tree** in which each node stores a unique key, and the keys are ordered.

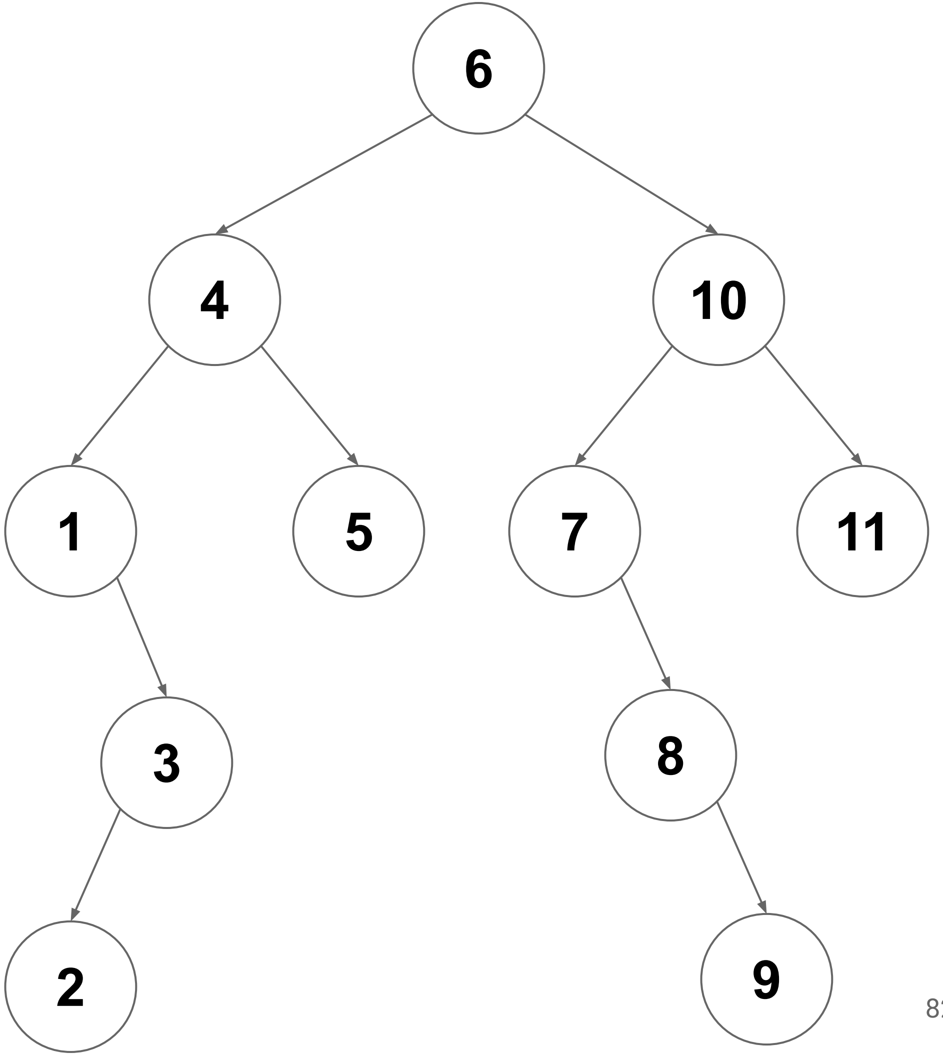
Constraints

- No duplicate keys
- For every node X_L in the left subtree of node X: X_L**.key** < X**.key**
- For every node X_R in the right subtree of node X: X_R**.key** > X**.key**

X **partitions** its children

Is this a valid BST?

Yes!



Finding an Item

Goal: Find an item with key ***k*** in a BST rooted at **`root`**

1. Is **`root`** empty? (if yes, then the item is not here)
2. Does **`root.value`** have key ***k***? (if yes, done!)
3. Is ***k*** less than **`root.value`**'s key? (if yes, search left subtree)
4. Is ***k*** greater than **`root.value`**'s key? (If yes, search the right subtree)

Inserting an Item

Goal: Insert a new item with key ***k*** in a BST rooted at **`root`**

1. Is **`root`** empty? (insert here)
2. Does **`root.value`** have key ***k***? (already present! don't insert)
3. Is ***k*** less than **`root.value`**'s key? (call insert on left subtree)
4. Is ***k*** greater than **`root.value`**'s key? (call insert on right subtree)

Removing an Item

Goal: Remove the item with key ***k*** from a BST rooted at **`root`**

1. **`find`** the item
2. Replace the found node with the right subtree
3. Insert the left subtree under the right

BST Operations

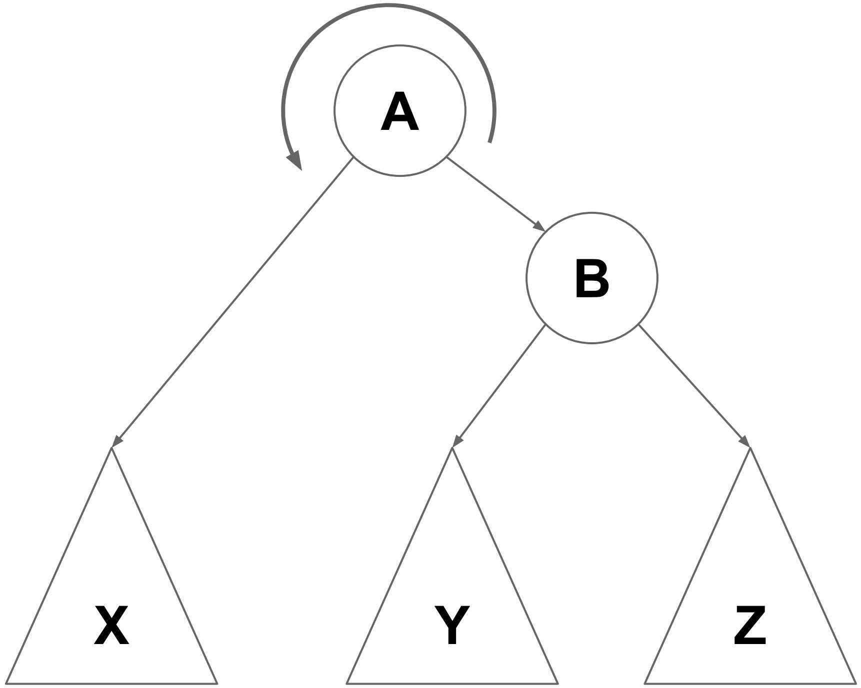
Operation	Runtime
`find`	***O*(*d*)**
`insert`	***O*(*d*)**
`remove`	***O*(*d*)**

*What is the runtime in terms of **n**?* ***O*(*n*)**

What about the lower bound? **Ω(log(*n*))**

Can we do better? **YES!**

Rebalancing Trees (rotations)



Rotate(A, B)

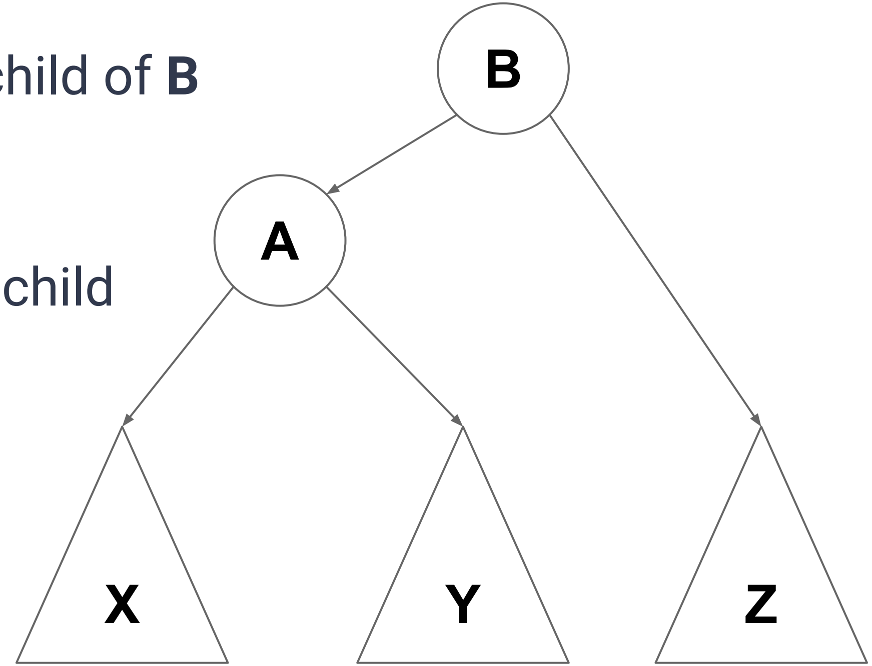
Rebalancing Trees (rotations)

Make **A** the left child of **B**

What about **Y**?

Make it the right child

of **A**



Rotate(A, B)

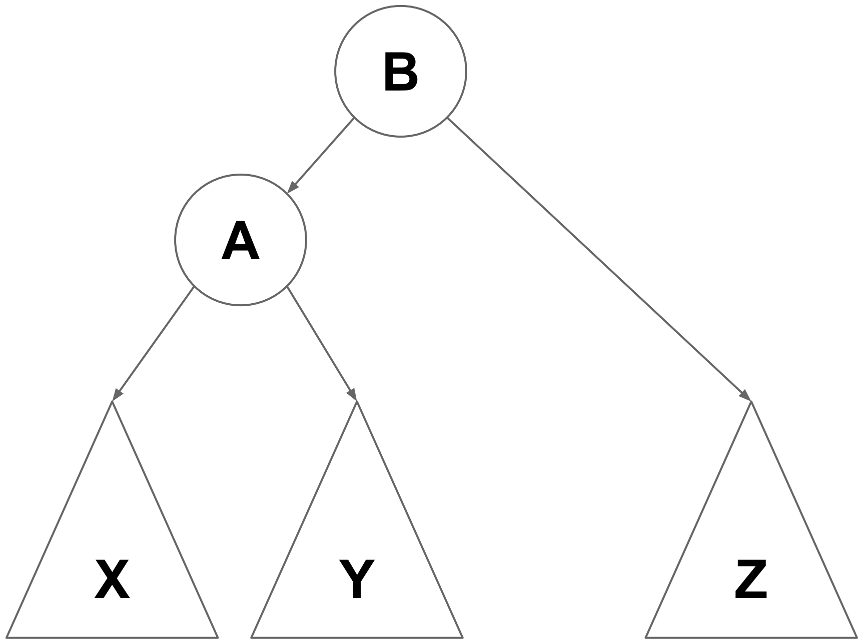
Rebalancing Trees (rotations)

A became **B**'s left child

B's left child became **A**'s right child

Is ordering maintained? **Yes!**

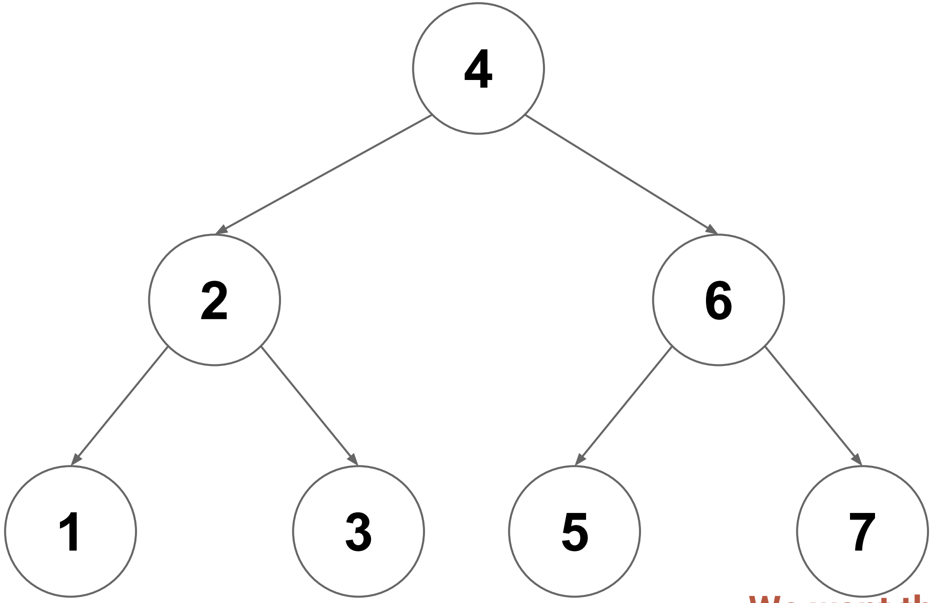
Complexity? ***O*(1)**



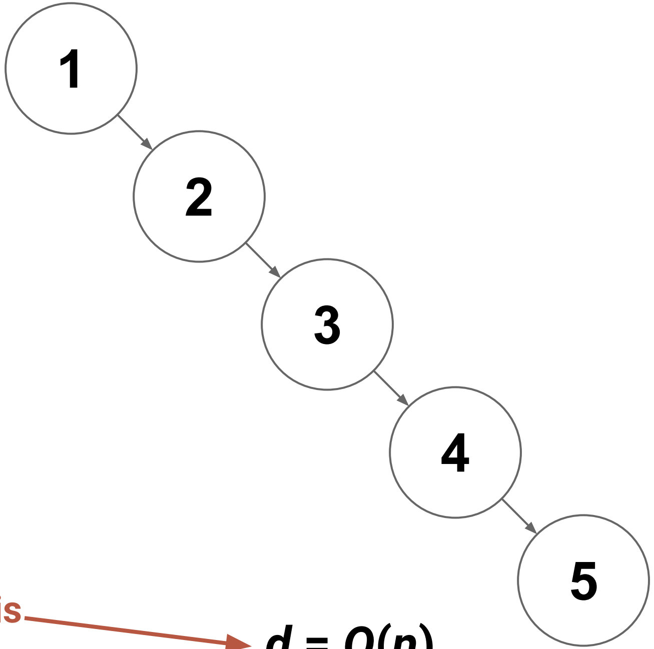
Rotate(A, B)

Tree Depth vs Size

If height(left) ≈ height(right)



If height(left) ≪ height(right)



We want this, not this

$d = O(\log(n))$

$d = O(n)$

AVL Trees

An **AVL tree** (**A**delson-**V**elsky and **L**andis) is a ***BST*** where every subtree is depth-balanced

Remember: Tree depth = height(root)

Balanced: |height(root.right) - height(root.left)| ≤ 1

AVL Trees - Depth Bounds

Question: Does the AVL property result in any guarantees about depth?

YES! Depth balance forces a maximum possible depth of **log(*n*)**

Proof Idea: An AVL tree with depth ***d*** has "enough" nodes

Inserting Records

To insert a record into an AVL Tree:

1. Find the insertion point (remember it is a BST) — $O(d) = O(\log n)$
2. Insert the new leaf and set balance factor to 0 — $O(1)$
3. Trace path back up to root and update balance factors — $O(d) = O(\log n)$
    a. If a balance factor becomes +/-2 then rotate to fix — $O(1)$

Removing Records

- Removal follows essentially the same process as insertion
  - Do a normal BST removal
  - Go back up the tree adjusting balance factors
  - If you discover a balance factor that goes to +2/-2, rotate to fix

Summary

- We want shallow BSTs (it makes **`find`**, **`insert`**, **`remove`** faster)
- Enforcing AVL constraints makes our BSTs shallow
  - The constraints are |height(right) - height(left)| ≤ 1
  - It will guarantee $d = O(\log(n))$
- Adding/removing from a BST changes height by at most 1
- A rotation can also change a BST height by at most 1
- Therefore after **`insert`**/**`remove`** into an AVL tree, we can reinforce AVL constraints with one (or two) rotations
  - We only need to make one trip back up the tree to do so
  - Therefore **`insert`**/**`remove`** is still $O(d) = O(\log(n))$

Maintaining Balance - Another Approach

Enforcing height-balance is too strict (May do “unnecessary” rotations)

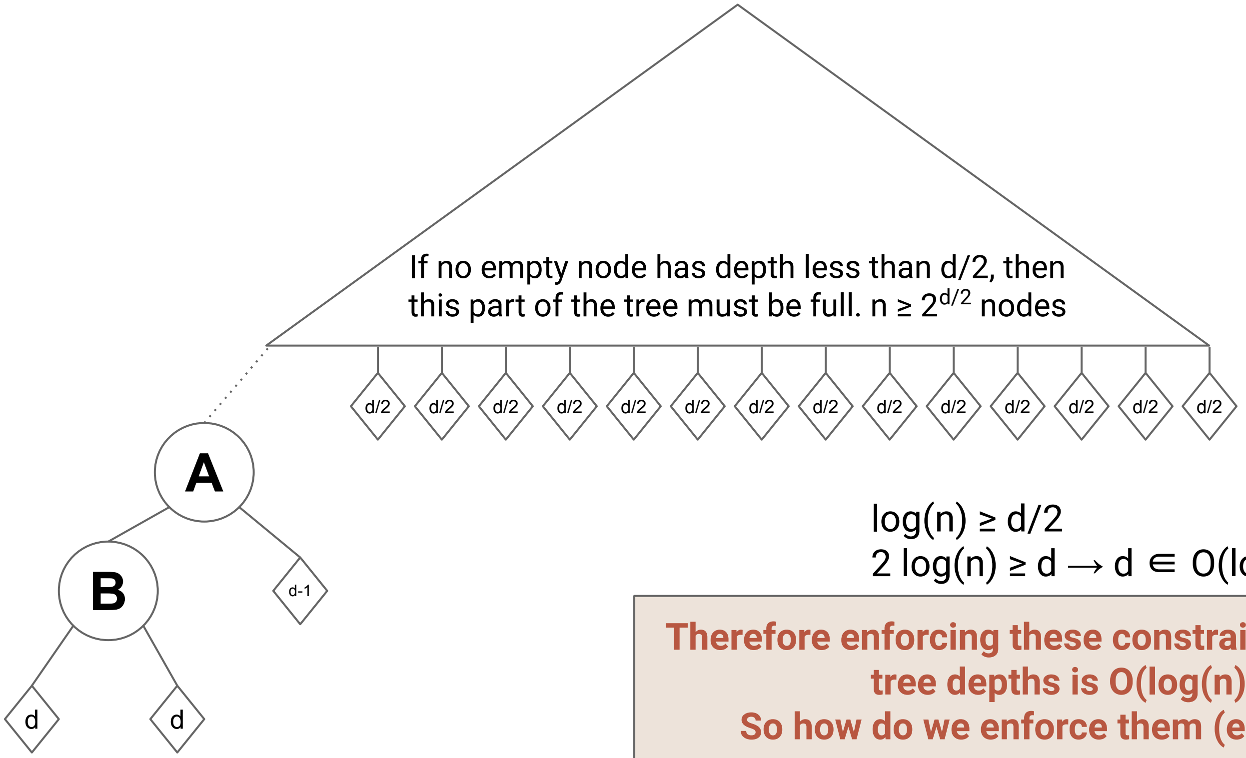
Weaker (and more direct) restriction:

- Balance the depth of empty tree nodes
- If ***a***, ***b*** are EmptyTree nodes, then enforce that for all ***a***, ***b***:
  - depth(***a***) ≥ (depth(***b***) ÷ 2)

    or

  - depth(***b***) ≥ (depth(***a***) ÷ 2)

Depth Balancing



$\log(n) \geq d/2$

$2\log(n) \geq d \rightarrow d \in O(\log(n))$

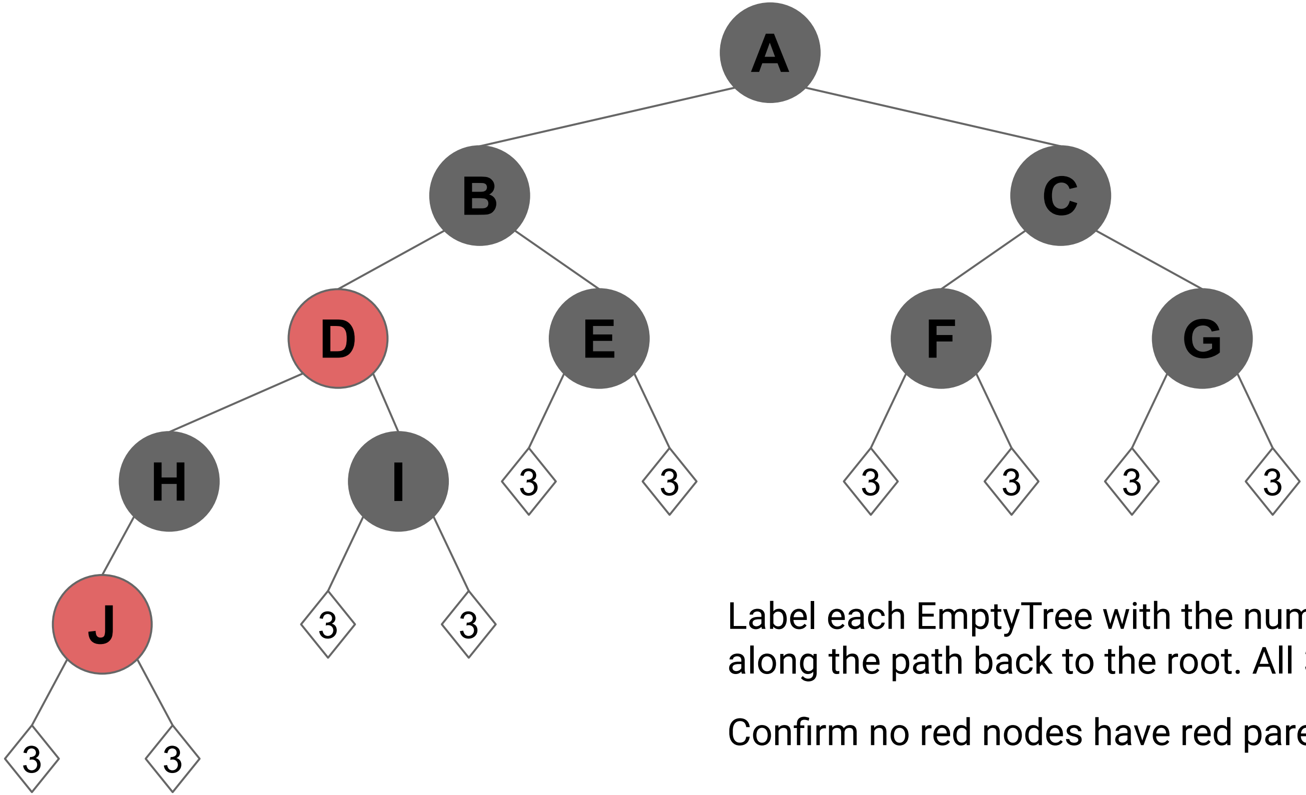
Therefore enforcing these constraints means that tree depths is O(log(n))...
So how do we enforce them (efficiently)?

Red-Black Trees

To Enforce the Depth Constraint on empty nodes:

1. Color each node red or black
    a. The # of black nodes from each empty node to root must be same
    b. The parent of a red node must always be black
2. On insertion (or deletion)
    a. Inserted nodes are red (won't break 1a)
    b. Repair violations of 1b by rotating and/or recoloring
        i. Make sure repairs don't break 1a

Red-Black Trees



Label each EmptyTree with the number of black nodes along the path back to the root. All 3 in this case ✓

Confirm no red nodes have red parents ✓

Red-Black Tree

Note: Each insertion creates at most one red-red parent-child conflict

- O(1) time to recolor/rotate to repair the parent-child conflict
- May create a red-red conflict in grandparent
  - Up to d/2 = O(log(n)) repairs required, but each repair is O(1)
- **Insertion therefore remains O(log(n))**

Note: Each deletion removes at most one black node (red doesn't matter)

- O(1) time to recolor/rotate to preserve black-depth
- May require recoloring (grand-)parent from black to red
  - Up to d = O(log(n)) repairs required
- **Deletion therefore remains O(log(n))**

BST Operations

Operation	BST	AVL	Red-Black
`find`	$O(d) = O(n)$	$O(d) = O(\log n)$	$O(d) = O(\log n)$
`insert`	$O(d) = O(n)$	$O(d) = O(\log n)$	$O(d) = O(\log n)$
`remove`	$O(d) = O(n)$	$O(d) = O(\log n)$	$O(d) = O(\log n)$

The tree operations on a BST are always $O(d)$ (they involve a constant number of trips from root to leaf at most).

The balanced varieties (AVL and Red-Black) constrain the depth

HashTables

Sets

A **Set** is an **unordered** collection of **unique** elements.

(order doesn't matter, and at most one copy of each ~~item~~ key)

The Set ADT

`void add(T element)`

Store one copy of **`element`** if not already present

`boolean contains(T element)`

Return true if **`element`** is present in the set

`boolean remove(T element)`

Remove **`element`** if present, or return false if not

Implementing Sets/Bags

	`add`	`contains`	`remove`
ArrayList	$O(n)$	$O(n)$	$O(n)$
LinkedList	$O(n)$	$O(n)$	$O(n)$
Sorted ArrayList	$O(n)$	$O(\log(n))$	$O(n)$
Sorted LinkedList	$O(n)$	$O(n)$	$O(n)$
General BST	$O(d) = O(n)$	$O(d) = O(n)$	$O(d) = O(n)$
Balanced BST	$O(d) = O(\log(n))$	$O(d) = O(\log(n))$	$O(d) = O(\log(n))$

Implementing Sets/Bags

	`add`	`contains`	`remove`
ArrayList	$O(n)$	$O(n)$	$O(n)$
LinkedList	$O(n)$	$O(n)$	$O(n)$
Sorted ArrayList	$O(n)$	$O(\log(n))$	$O(n)$
Sorted LinkedList	***Can we improve on this even further?***		
General BST	$O(d) = O(n)$	$O(d) = O(n)$	$O(d) = O(n)$
Balanced BST	$O(d) = O(\log(n))$	$O(d) = O(\log(n))$	$O(d) = O(\log(n))$

Finding Items

When implementing these operations with a BST where is most of "cost" of each algorithm coming from? **Finding the element**

contains => **find the element**
add => **find the insertion point**, then add (the add is often O(1))
remove => **find the element**, then remove (the remove is often O(1))

What if we could just...skip the find step?
What if we knew exactly where the element would be?

Assigning Bins

Which data structure has constant lookup if we know where our element is in a sequence? **An Array**

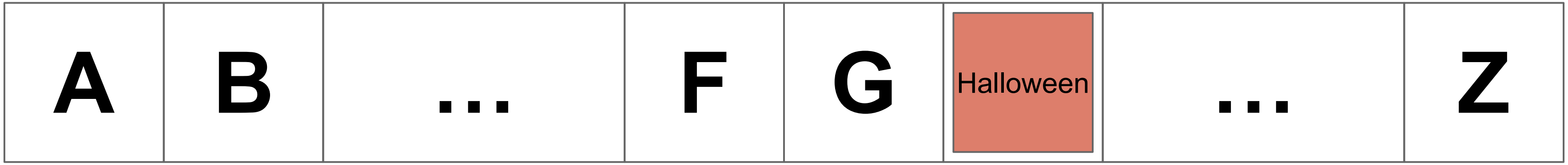
Idea: What if we could assign each record to a location in an Array

- Create and array of size ***N***
- Pick an ***O*(1)** function to assign each record a number in **[0,*N*)**
  - ie: creating a set of movies stored by first letter of title, String →[0,26)

Assigning Bins

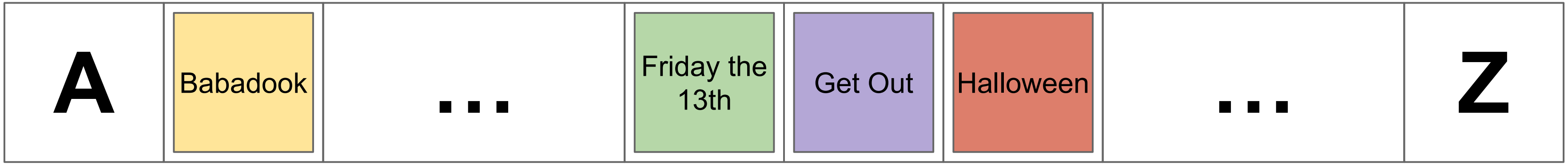
`add("Halloween")` → `"Halloween"[0] == "H" == 7`

This computation is ***O*(1)**



Assigning Bins

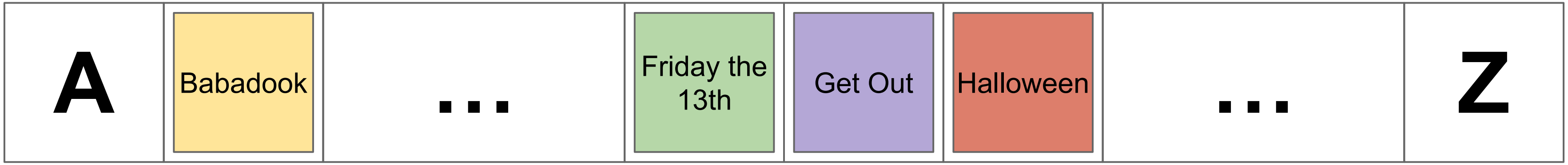
`add("Babadook")` → `"Babadook"[0] == "B" == 1`



Assigning Bins

`contains("Get Out") → "Get Out"[0] == "G" == 6`

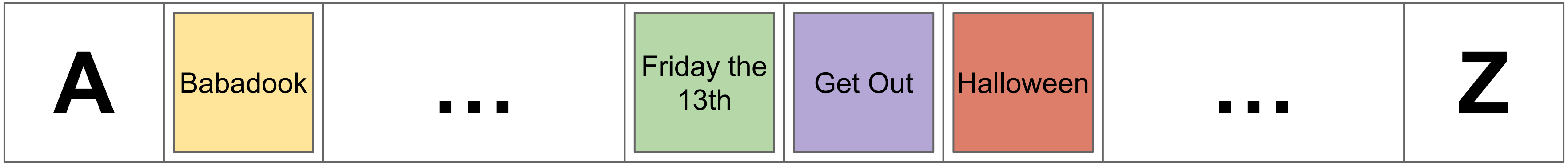
Find in constant time!



Assigning Bins

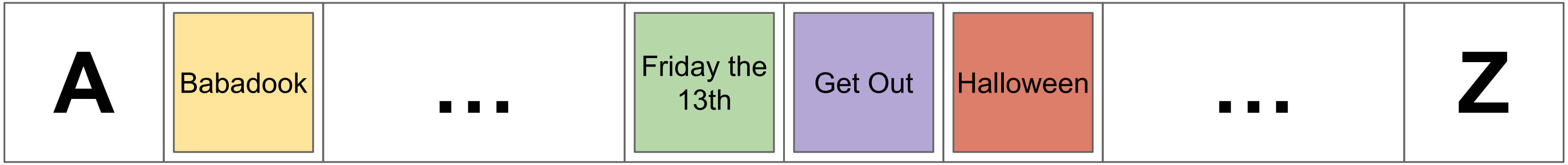
`contains("Scream")` → `"Scream"[0] == "S" == 18`

Determine that "Scream" is not in the Set in constant time!



Assigning Bins

What about: `contains("Hereditary")`?



Once we know the location, we still need to check for an exact match.

`"Hereditary"[0] == "H" == 7, Array[7] != "Hereditary"`

Determine that "Hereditary" is not in the Set in constant time!

Assigning Bins

Pros

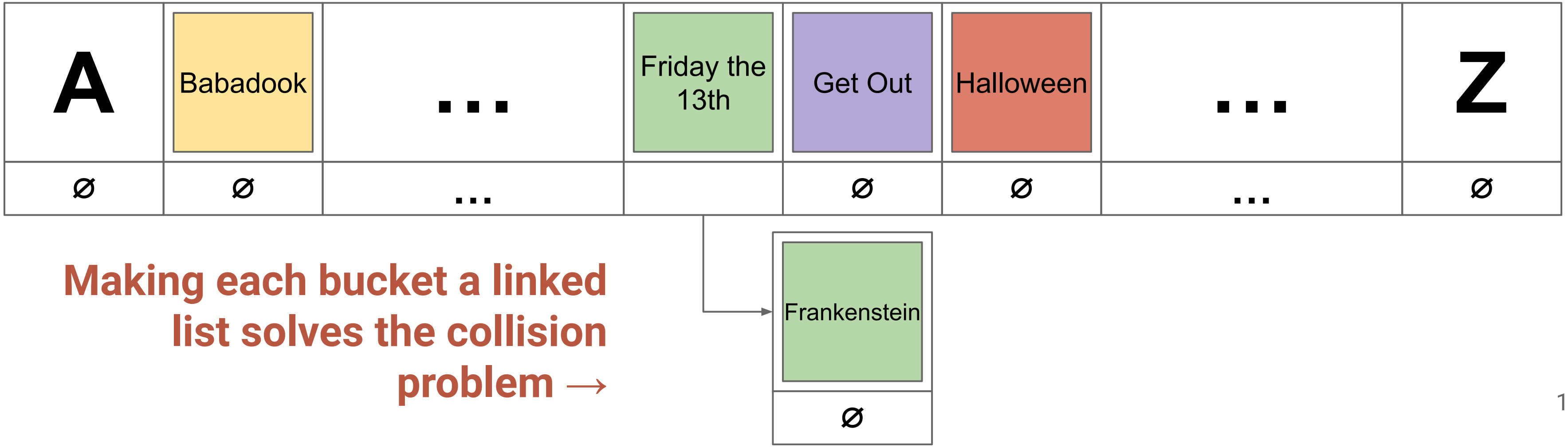
- ***O*(1)** insert
- ***O*(1)** find
- ***O*(1)** remove

Cons

- Wasted space (4/26 slots used in the example, will we ever use "Z"?)
- Duplication (What about inserting **F**rankenstein)

Assigning Bins

`add("Frankenstein")?`



Making each bucket a linked list solves the collision problem →

LinkedList Bins

Now we can handle as many duplicates as we need. But are we losing our constant time operations?

How many elements are we expecting to end up in each bucket?

Depends partially on our choice of Hash Function

Picking a Hash Function

Desirable features for *h*(*x*):

- Fast — needs to be $O(1)$
- "Unique" – As few duplicate bins as possible

Hash Functions In the Real-World

Examples

- SHA256 ← Used by GIT
- MD5, BCRYPT ← Used by unix login, apt
- MurmurHash3 ← Used by Scala

hash(*x*) is pseudo-random

- **hash(*x*)** ~ uniform random value in [0, INT_MAX)
- **hash(*x*)** always returns the same value for the same ***x***
- **hash(*x*)** is uncorrelated with **hash(*y*)** for all x ≠ y

Refresher on Modulus

The modulus function takes any integers ***n*** and ***d***, and returns a number ***r*** in the range [0, ***d***), such that ***n*** = ***q * d + r***. (It returns the remainder of ***n / d***)

0	**1**	**2**	**3**	**4**	**5**	**6**

If my hash table has 7 buckets, and I insert an element with hash code 73, what bucket would it go in? **73 % 7 = 3**

Pseudo-Random Hash Function

$$n = \text{number of elements in any bucket}$$

$$N = \text{number of buckets}$$

$$b_{i,j} = \begin{cases} 1 & \textbf{if } \text{element } i \text{ is assigned to bucket } j \\ 0 & otherwise \end{cases}$$

$$\mathbb{E}\left[b_{i,j}\right] = \frac{1}{N}$$

Pseudo-Random Hash Function

$$n = \text{number of elements in any bucket}$$

$$N = \text{number of buckets}$$

$$b_{i,j} = \begin{cases} 1 & \textbf{if } \text{element } i \text{ is assigned to bucket } j \\ 0 & otherwise \end{cases}$$

$$\mathbb{E}\left[\sum_{i=0}^{n} b_{i,j}\right] = \frac{n}{N}$$

Pseudo-Random Hash Function

$$n = \text{number of elements in any bucket}$$

$$N = \text{number of buckets}$$

$$b_{i,j} = \begin{cases} 1 & \textbf{if } \text{element } i \text{ is assigned to bucket } j \\ 0 & otherwise \end{cases}$$

Only true if $b_{i,j}$ and $b_{i',j}$ are uncorrelated for any i ≠ i'

(h(i) can't be related to h(i'))

$$\mathbb{E}\left[\sum_{i=0}^{n} b_{i,j}\right] = \frac{n}{N}$$

The **expected** number of elements in any bucket j

Pseudo-Random Hash Function

$$n = \text{number of elements in any bucket}$$

$$N = \text{number of buckets}$$

$$b_{i,j} = \begin{cases} 1 & \textbf{if } \text{element } i \text{ is assigned to bucket } j \\ 0 & \textit{otherwise} \end{cases}$$

Only true if $b_{i,j}$ and $b_{i',j}$ are uncorrelated for any i ≠ i'

(h(i) can't be related to h(i'))

$$\mathbb{E}\left[\sum_{i=0}^{n} b_{i,j}\right] = \frac{n}{N}$$

The **expected** number of elements in any bucket j

…given this information, what do the runtimes of our operations look like?

Pseudo-Random Hash Function

$$n = \text{number of elements in any bucket}$$

$$N = \text{number of buckets}$$

$$b_{i,j} = \begin{cases} 1 & \textbf{if} \text{ element } i \text{ is assigned to bucket } j \\ 0 & \textit{otherwise} \end{cases}$$

Expected runtime of `insert`, `apply`, `remove`: ***O*(*n*/*N*)**

Worst-Case runtime of `insert`, `apply`, `remove`: ***O*(*n*)**

Hash Functions + Buckets

Everything is: $O\left(\frac{n}{N}\right)$ Let's call $\alpha = \frac{n}{N}$ the load factor.

Idea: Make α a constant

Fix an α_{max} and start requiring that $\alpha \leq \alpha_{max}$

What do we do when this constraint is violated? **Resize!**

Hash Function Recap

- We now have ***pseudo-random*** hash functions that run in ***O***(1)
  - They act as if they are uniformly random
    - Will evenly distribute elements to buckets
    - hash(***x***) is uncorrelated with hash(***y***)
  - They are deterministic (hash(***x***) will always return the same value)
- We can use these hash functions to determine which bucket an arbitrary element belongs in in ***O***(1) time
- There are expected to be ***n/N*** elements in that bucket
  - So runtime for all operations is **expected *O*(1) + *O*(*n/N*)**

Next goal: Make this a constant

Rehashing

When we insert an element that would exceed the load factor we:

1. Resize the underlying array from N_{old} to N_{new}
2. Rehash all of the elements from their old bucket to their new bucket
   a. Element x moves from hash(x) % N_{old} to hash(x) % N_{new}

How long does this take?

1. Allocate the new array: $O(1)$
2. Rehash every element from the old array to the new: $O(N_{old} + n)$
3. Free the old array: $O(1)$

Total: $O(N_{old} + n)$

Recap of HashTables (so far...)

Current Design: HashTable with Chaining

- Array of buckets
- Each bucket is the head of a linked list (a "chain" of elements)

Runtime for `apply(x)`

<u>Expected Runtime:</u>

1. Find the bucket (call our hash function): $\boldsymbol{O(c_{hash}) = O(1)}$
2. Find the record in the bucket: $\boldsymbol{O(\alpha \cdot c_{equality}) = O(1)}$
3. **Total:** $\boldsymbol{O(c_{hash} + \alpha \cdot c_{equality}) = O(1)}$

<u>Unqualified Worst-Case:</u>

1. Find the bucket (call our hash function): $\boldsymbol{O(c_{hash}) = O(1)}$
2. Find the record in the bucket: $\boldsymbol{O(n \cdot c_{equality}) = O(n)}$
3. **Total:** $\boldsymbol{O(c_{hash} + n \cdot c_{equality}) = O(n)}$

Runtime for `remove(x)`

Expected Runtime:

1. Find the bucket (call our hash function): $O(c_{hash}) = O(1)$
2. Find the record in the bucket: $O(\alpha \cdot c_{equality}) = O(1)$
3. Remove (by reference): $O(1)$
4. **Total:** $O(c_{hash} + \alpha \cdot c_{equality} + 1) = O(1)$

Only one extra constant-time step to remove

Unqualified Worst-Case:

1. Find the record in the bucket: $O(n \cdot c_{equality}) = O(n)$
2. **Total:** $O(c_{hash} + n \cdot c_{equality} + 1) = O(n)$

Runtime for `insert(x)`

Expected Runtime:

1. Find the bucket (call our hash function): $O(c_{hash}) = O(1)$
2. Remove ***x*** from bucket if present: $O(\alpha \cdot c_{equality} + 1)$
3. Prepend to bucket: $O(1)$
4. Rehash if needed: $O(n \cdot c_{hash} + N)$ (amortized $O(1)$)
5. **Total:** $O(c_{hash} + \alpha \cdot c_{equality} + 3) = O(1)$

One additional constant-time step to prepend, and then potentially the need to rehash, but that is amortized O(1)

Unqualified Worst-Case:

1. Remove ***x*** from bucket if present: $O(n \cdot c_{equality} + 1) = O(n)$
2. **Total:** $O(c_{hash} + n \cdot c_{equality} + 3) = O(n)$

HashTables with Chaining

hash(A) = 4

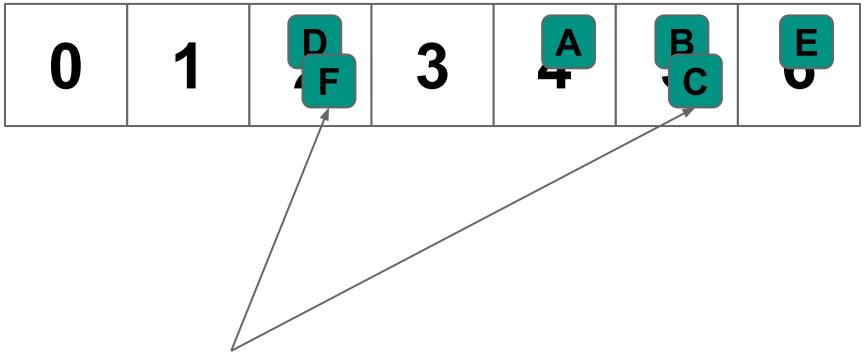
hash(B) = 5

hash(C) = 5

hash(D) = 2

hash(E) = 6

hash(F) = 2



Collisions are resolved by adding the element to the buckets linked list

HashTables with Open Addressing

hash(A) = 4 ← no collision

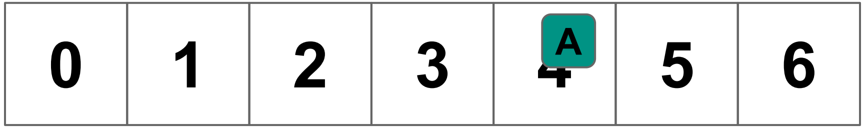
hash(B) = 5

hash(C) = 5

hash(D) = 2

hash(E) = 6

hash(F) = 4



With Open Addressing collisions are resolved by "cascading" to the next available bucket

HashTables with Open Addressing

hash(A) = 4

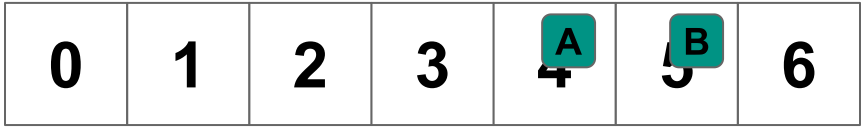
hash(B) = 5 ← no collision

hash(C) = 5

hash(D) = 2

hash(E) = 6

hash(F) = 4



With Open Addressing collisions are resolved by "cascading" to the next available bucket

HashTables with Open Addressing

hash(A) = 4

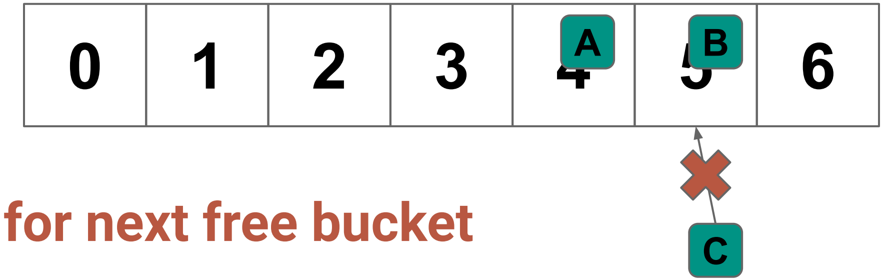
hash(B) = 5

hash(C) = 5 ← collision! Search for next free bucket

hash(D) = 2

hash(E) = 6

hash(F) = 4



With Open Addressing collisions are resolved by "cascading" to the next available bucket

HashTables with Open Addressing

hash(A) = 4

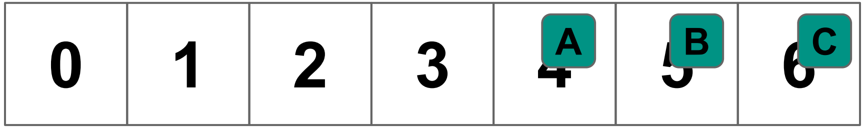
hash(B) = 5

hash(C) = 5 ← collision! Search for next free bucket

hash(D) = 2

hash(E) = 6

hash(F) = 4



With Open Addressing collisions are resolved by "cascading" to the next available bucket

HashTables with Open Addressing

hash(A) = 4

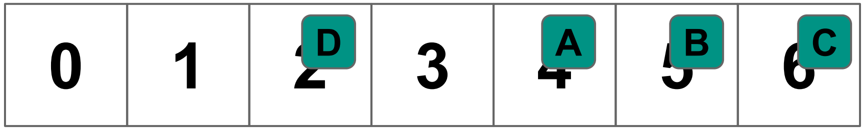
hash(B) = 5

hash(C) = 5

hash(D) = 2 ← no collision!

hash(E) = 6

hash(F) = 4



With Open Addressing collisions are resolved by "cascading" to the next available bucket

HashTables with Open Addressing

hash(A) = 4

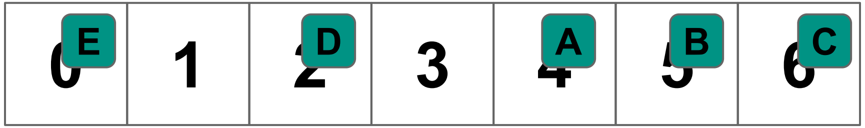
hash(B) = 5

hash(C) = 5

hash(D) = 2

hash(E) = 6 ← collision! cascade to 0

hash(F) = 4



With Open Addressing collisions are resolved by "cascading" to the next available bucket

HashTables with Open Addressing

hash(A) = 4

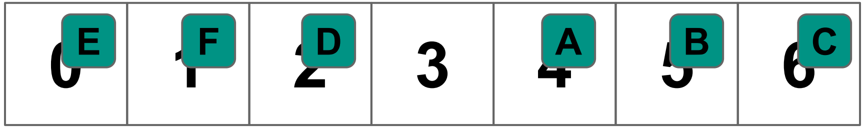
hash(B) = 5

hash(C) = 5

hash(D) = 2

hash(E) = 6

hash(F) = 4 ← collision! Cascade all the way to 1



With Open Addressing collisions are resolved by "cascading" to the next available bucket

Cuckoo Hashing

Idea: Use two hash functions, $hash_1$ and $hash_2$

To insert a record ***X***:

1. If $hash_1$(***X***) and $hash_2$(***X***) are both available, pick one at random
2. If only one of those buckets is available, pick the available bucket
3. If neither is available, pick one at random and evict the record there
   a. Insert ***X*** in this bucket
   b. Insert the evicted record following the same procedure

HashTables with Cuckoo Hashing

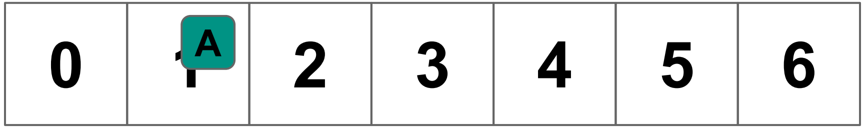
$hash_1(A) = 1$ $hash_2(A) = 3$

$hash_1(B) = 2$ $hash_2(B) = 4$

$hash_1(C) = 2$ $hash_2(C) = 1$

$hash_1(D) = 4$ $hash_2(D) = 6$

$hash_1(E) = 3$ $hash_2(E) = 4$



HashTables with Cuckoo Hashing

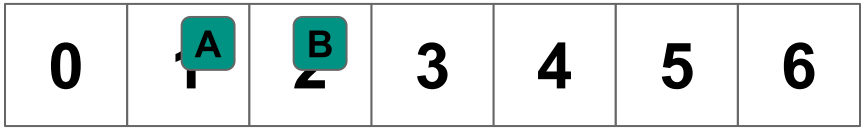
$hash_1(A) = 1$ $hash_2(A) = 3$

$hash_1(B) = 2$ $hash_2(B) = 4$

$hash_1(C) = 2$ $hash_2(C) = 1$

$hash_1(D) = 4$ $hash_2(D) = 6$

$hash_1(E) = 3$ $hash_2(E) = 4$



HashTables with Cuckoo Hashing

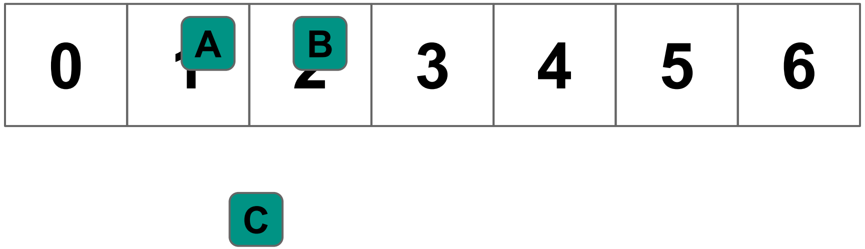
$hash_1(A) = 1$ $hash_2(A) = 3$

$hash_1(B) = 2$ $hash_2(B) = 4$

$hash_1(C) = 2$ **$hash_2(C) = 1$**

$hash_1(D) = 4$ $hash_2(D) = 6$

$hash_1(E) = 3$ $hash_2(E) = 4$



C can't go in either bucket, so evict one at random (let's say ***B***) and reinsert the evicted element

HashTables with Cuckoo Hashing

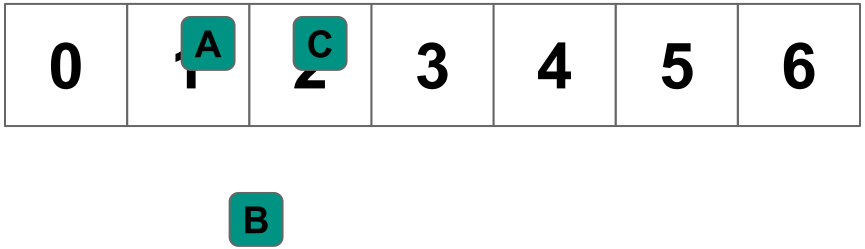
$hash_1(A) = 1$ $hash_2(A) = 3$

$hash_1(B) = 2$ **$hash_2(B) = 4$**

$hash_1(C) = 2$ $hash_2(C) = 1$

$hash_1(D) = 4$ $hash_2(D) = 6$

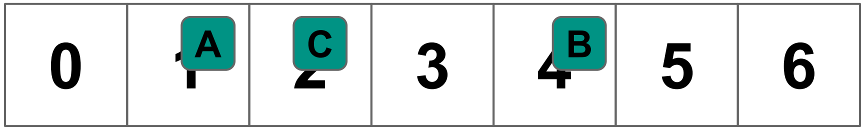
$hash_1(E) = 3$ $hash_2(E) = 4$



B can only go in 4 now, but 4 is free

HashTables with Cuckoo Hashing

$hash_1(A) = 1$	$hash_2(A) = 3$
$hash_1(B) = 2$	$hash_2(B) = 4$
$hash_1(C) = 2$	$hash_2(C) = 1$
$hash_1(D) = 4$	$hash_2(D) = 6$
$hash_1(E) = 3$	$hash_2(E) = 4$



B can only go in 4 now, but 4 is free

HashTables with Cuckoo Hashing

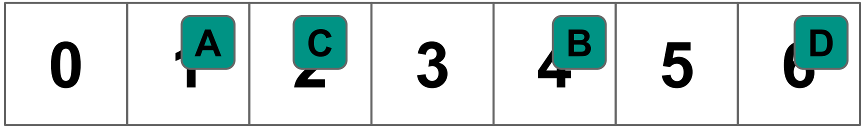
$hash_1(A) = 1$ $hash_2(A) = 3$

$hash_1(B) = 2$ $hash_2(B) = 4$

$hash_1(C) = 2$ $hash_2(C) = 1$

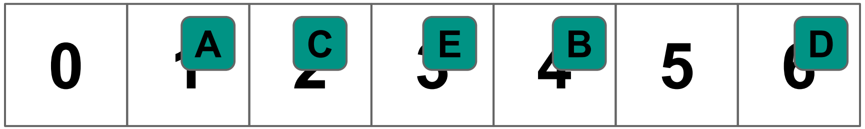
$hash_1(D) = 4$ **$hash_2(D) = 6$**

$hash_1(E) = 3$ $hash_2(E) = 4$



HashTables with Cuckoo Hashing

$hash_1(A) = 1$	$hash_2(A) = 3$
$hash_1(B) = 2$	$hash_2(B) = 4$
$hash_1(C) = 2$	$hash_2(C) = 1$
$hash_1(D) = 4$	$hash_2(D) = 6$
$hash_1(E) = 3$	$hash_2(E) = 4$



HashTables with Cuckoo Hashing

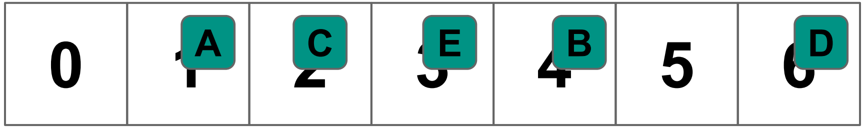
$hash_1(A) = 1$ $hash_2(A) = 3$

$hash_1(B) = 2$ $hash_2(B) = 4$

$hash_1(C) = 2$ $hash_2(C) = 1$

$hash_1(D) = 4$ $hash_2(D) = 6$

$\mathbf{hash_1(E) = 3}$ $hash_2(E) = 4$



What if we try to insert ***F*** which hashes to either 1 or 3?

HashTables with Cuckoo Hashing

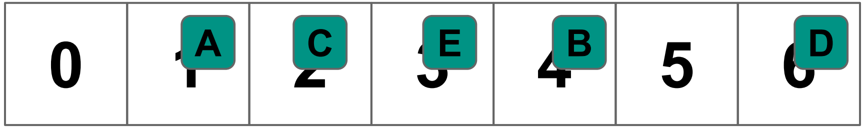
$hash_1(A) = 1$ $hash_2(A) = 3$

$hash_1(B) = 2$ $hash_2(B) = 4$

$hash_1(C) = 2$ $hash_2(C) = 1$

$hash_1(D) = 4$ $hash_2(D) = 6$

$\mathbf{hash_1(E) = 3}$ $hash_2(E) = 4$



What if we try to insert ***F*** which hashes to either 1 or 3? **We will loop infinitely trying to evict…so limit the number of eviction attempts then do a full rehash**

Cuckoo Hashing

So with Cuckoo Hashing, we may have to rehash early, and may follow long chains of evictions inserting, but…

What is the runtime of apply/remove?

Cuckoo Hashing

So with Cuckoo Hashing, we may have to rehash early, and may follow long chains of evictions inserting, but…

What is the runtime of apply/remove?

1. Check 2 different buckets: ***O*(1)**
2. That's it…no chaining, cascading etc…

Apply and remove are <u>GUARANTEED</u> *O*(1) with Cuckoo Hashing

Implementing Sets/Bags

	`add`	`contains`	`remove`
ArrayList	$O(n)$	$O(n)$	$O(n)$
LinkedList	$O(n)$	$O(n)$	$O(n)$
Sorted ArrayList	$O(n)$	$O(\log(n))$	$O(n)$
Sorted LinkedList	$O(n)$	$O(n)$	$O(n)$
General BST	$O(d) = O(n)$	$O(d) = O(n)$	$O(d) = O(n)$
Balanced BST	$O(d) = O(\log(n))$	$O(d) = O(\log(n))$	$O(d) = O(\log(n))$
HashTable	*expected* $O(1)$	*expected* $O(1)$	*expected* $O(1)$

HashTable Drawbacks?

…So the expected runtime of all operations is ***O*(1)**

Why would you ever use any other data structure?

- HashTables do not preserve ordering
- HashTables may waste a lot of memory
- Rehashing can be expensive
- Only **guarantee** on lookup time is that it is ***O*(*n*)**

Misc Topics

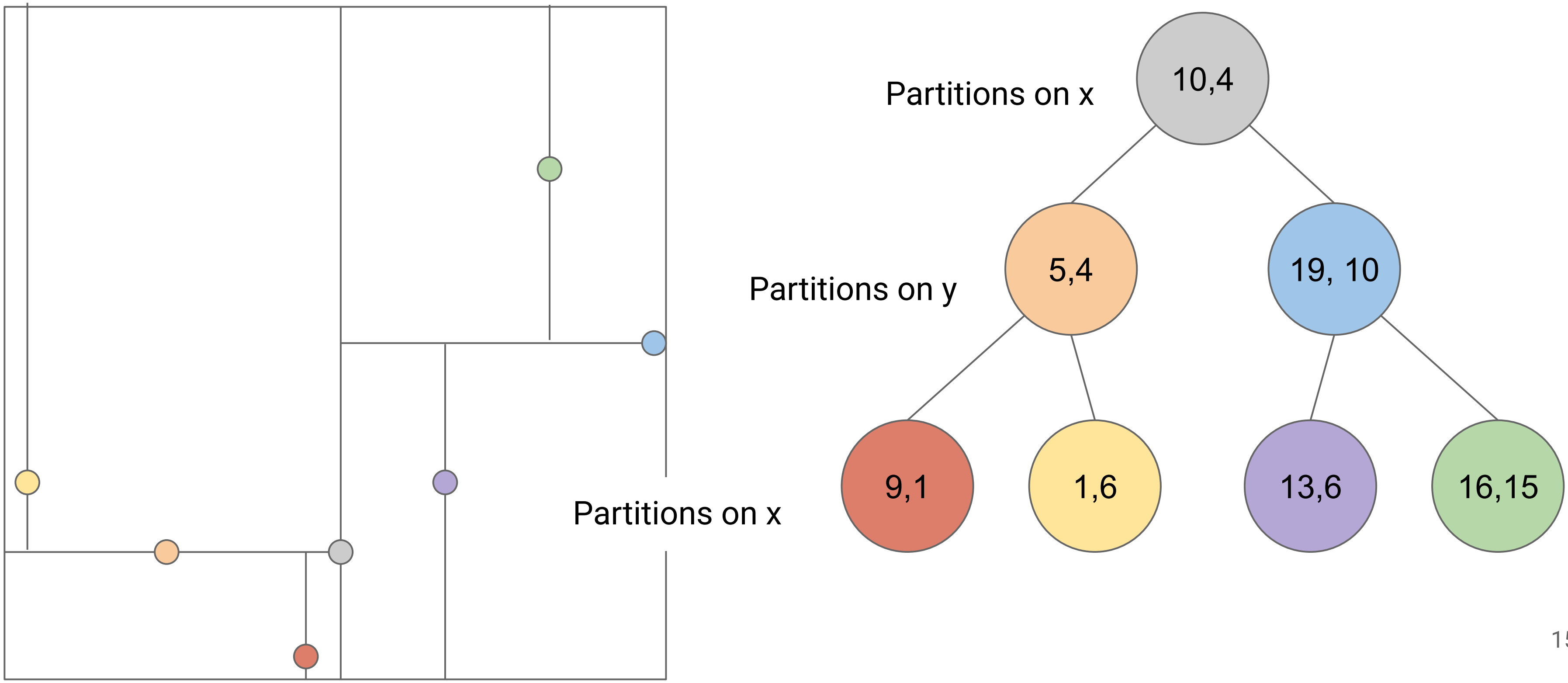
k-D Trees

- Can generalize to k>2 dimensions
  - Depth 0: Partition on Dimension 0
  - Depth 1: Partition on Dimension 1
  - …
  - Depth k-1: Partition on Dimension k-1
  - Depth k: Partition on Dimension 0
  - Depth k+1: Partition on Dimension 1
  - Depth i: Partition on Dimension (i mod k)

The name k-D tree comes from this generalization (k-Dimensional Tree)

- In practice, `range()` and `knn()` become ~ ***O*(*n*)** for k > 3
  - If a subtree's range overlaps with the target in even one dimension, we need to search it. (Curse of Dimensionality)

k-D Tree


Partitions on x
10,4
Partitions on y
5,4
19, 10
Partitions on x
9,1
1,6
13,6
16,15

Quad/Oct Trees Revisited

Idea: Let's organize the data (spatially) in a tree structure

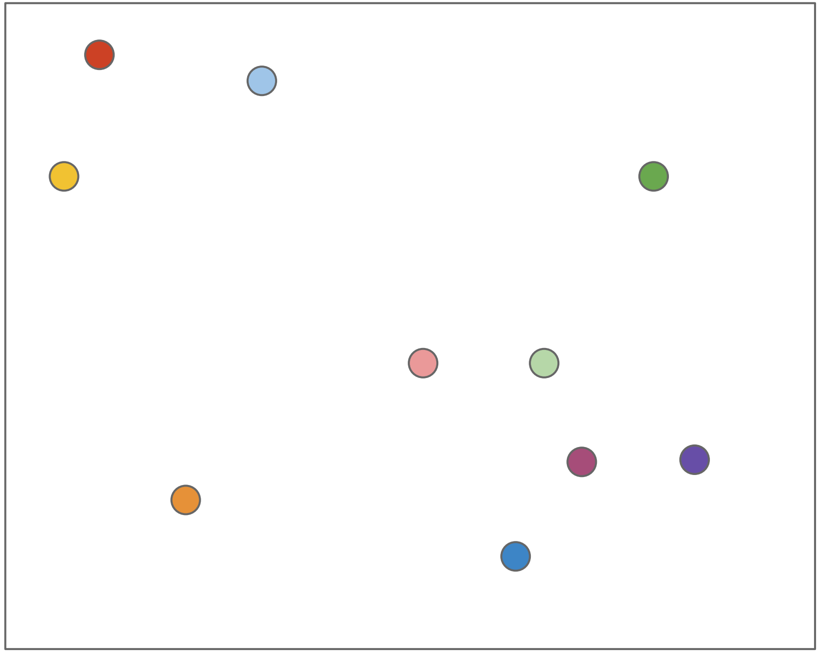
- 2D space → use a quad tree
- 3D space → use an oct tree (each node has at most 8 children)

Unlike last time, let's partition the space we are simulating, rather than the points in the space

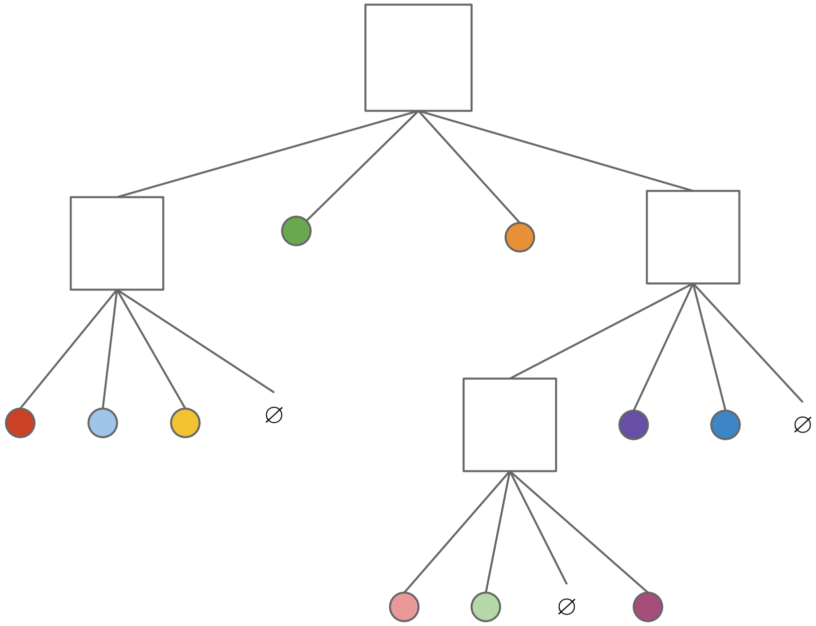
Space Partitioning - 2D Example

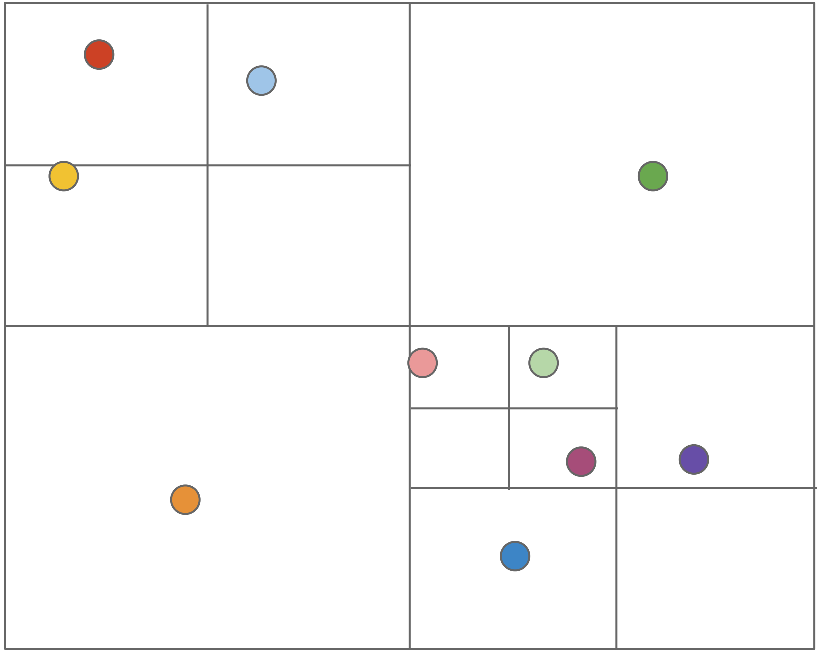
Create a quad-tree by recursively partitioning the space

- Divide the space evenly until there is only one element per partition
- Internal tree nodes represent the partitions, leaves are the actual elements



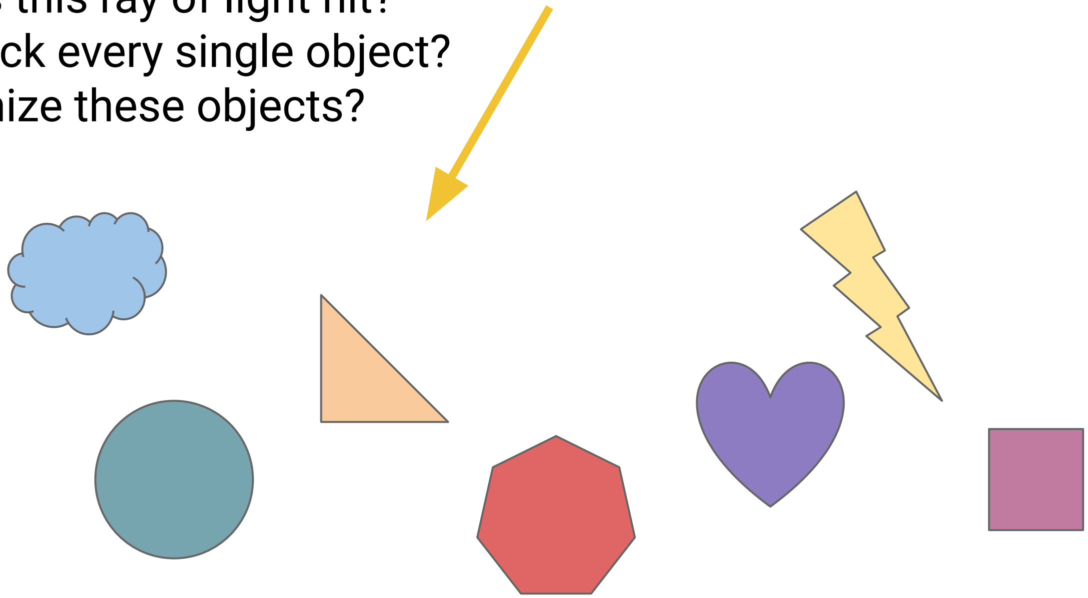
Space Partitioning - 2D Example





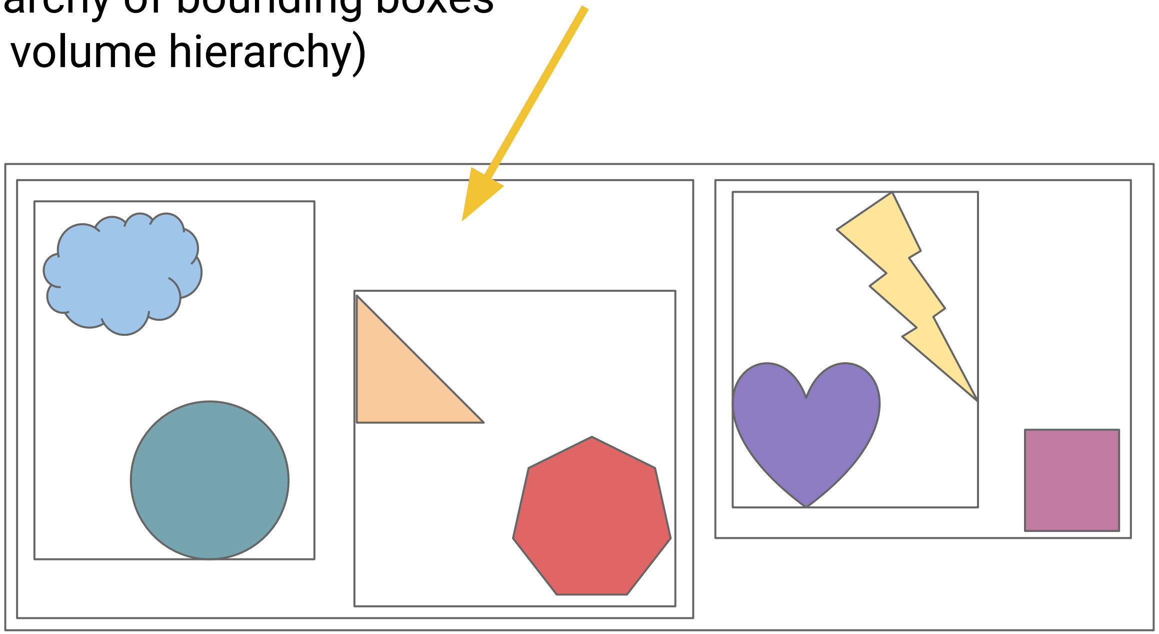
Other Problems: Ray/Path Tracing

Which object does this ray of light hit?
Do we have to check every single object?
How can we organize these objects?



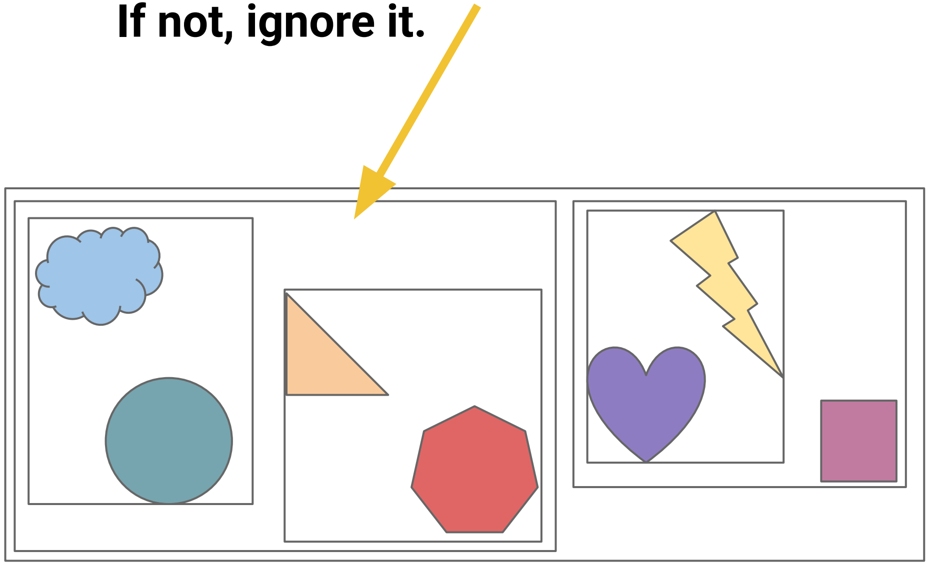
Other Problems: Ray/Path Tracing

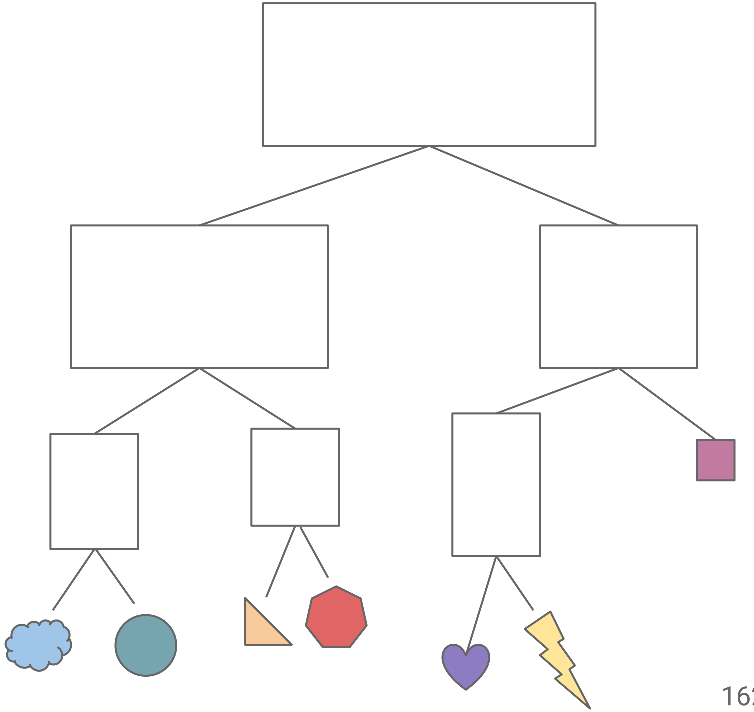
Idea: Build a hierarchy of bounding boxes
(BVH - Bounding volume hierarchy)



Other Problems: Ray/Path Tracing

These bounding boxes form a tree…
We can check if the ray intersects a bounding box.
If it does, explore its children.
If not, ignore it.





High-Level Summary

- We've seen both trees and hash tables as effective ways to organize our data if we know we are going to be searching it often
- **HashTables** can be great for exact lookups
  - Think PA3: you may want to lookup a person with an exact (birthday, zipcode) pair, and HashTable lets you do that very fast
- **Trees** and tree like structures work very well for "fuzzier" searches
  - What is "close" to this point? What object might this projectile hit? etc
  - The input to your search is not necessarily an exact element in your tree, but the tree organizes the data in a way that directs your search

Algorithmic Complexity

Remember: $\boldsymbol{O(f(n))}$ placed bounds on *growth functions* in general. Not necessarily only for runtime growth functions…

Runtime Bounds (or Runtime Complexity)

- The algorithm takes $\boldsymbol{O}$(...) time

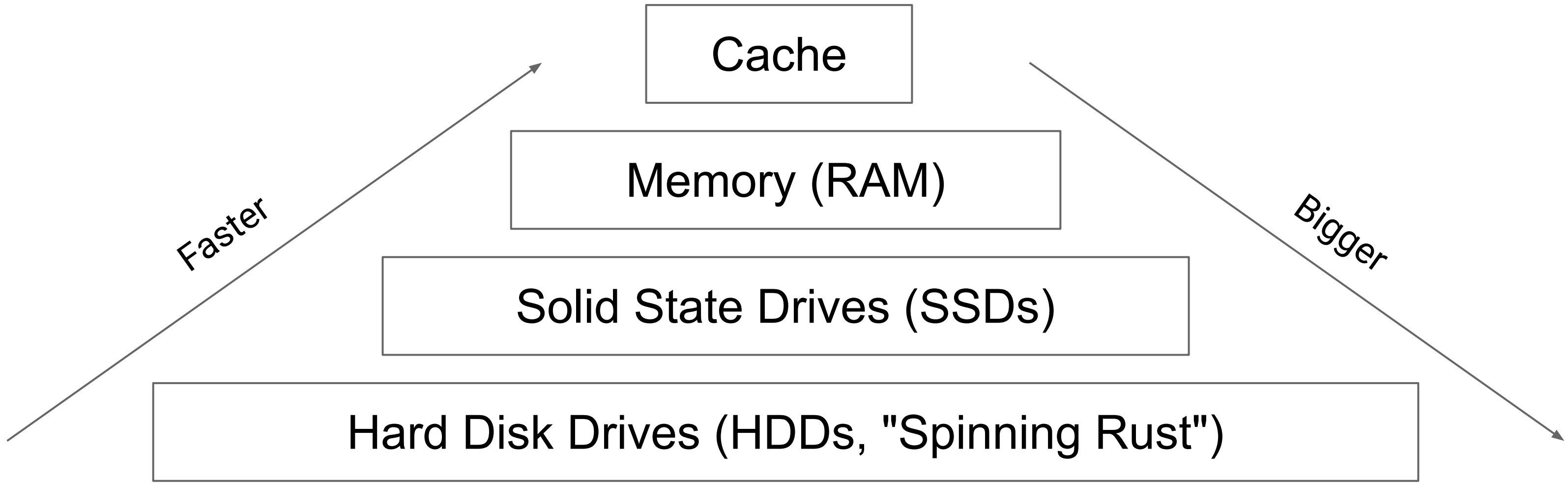
Memory Bounds (or Memory Complexity)

- The algorithm needs $\boldsymbol{O}$(...) storage

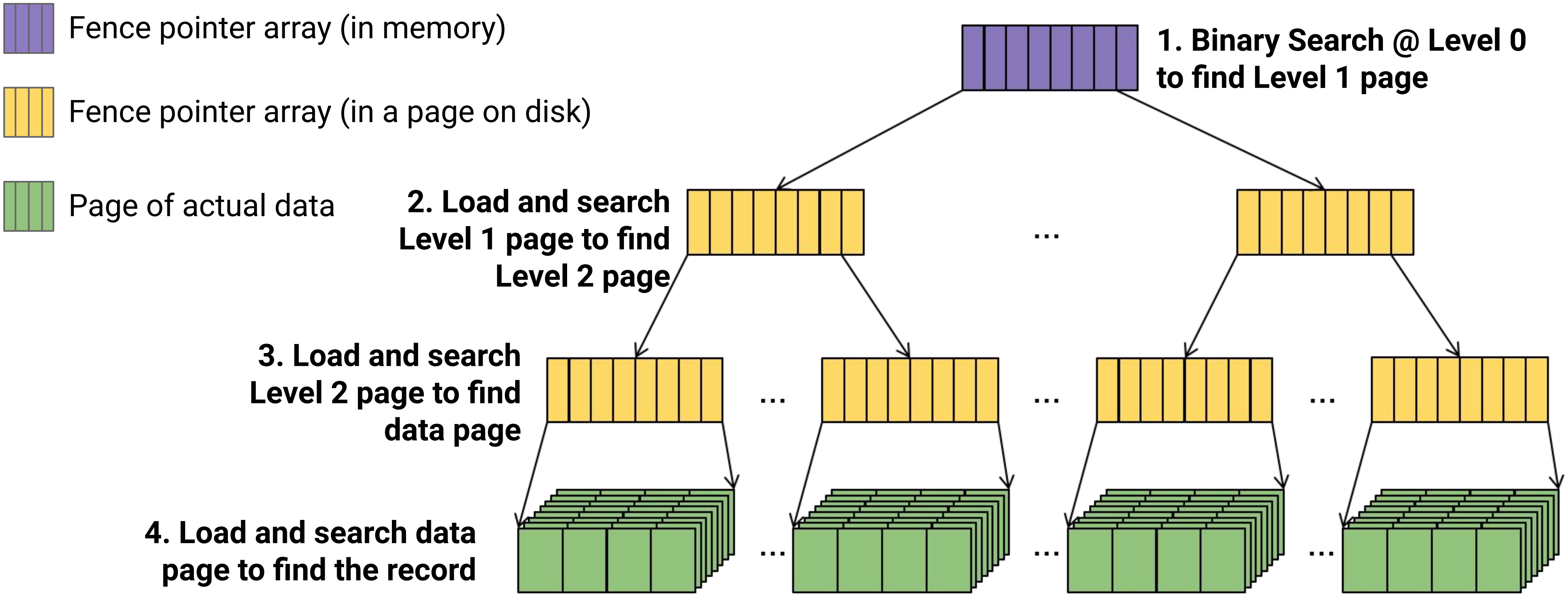
I/O Bounds (or I/O Complexity)

- The algorithm performs $\boldsymbol{O}$(...) accesses to slower memory

The Memory Hierarchy (simplified)



~~Improving on Fence Pointers~~ ISAM Index



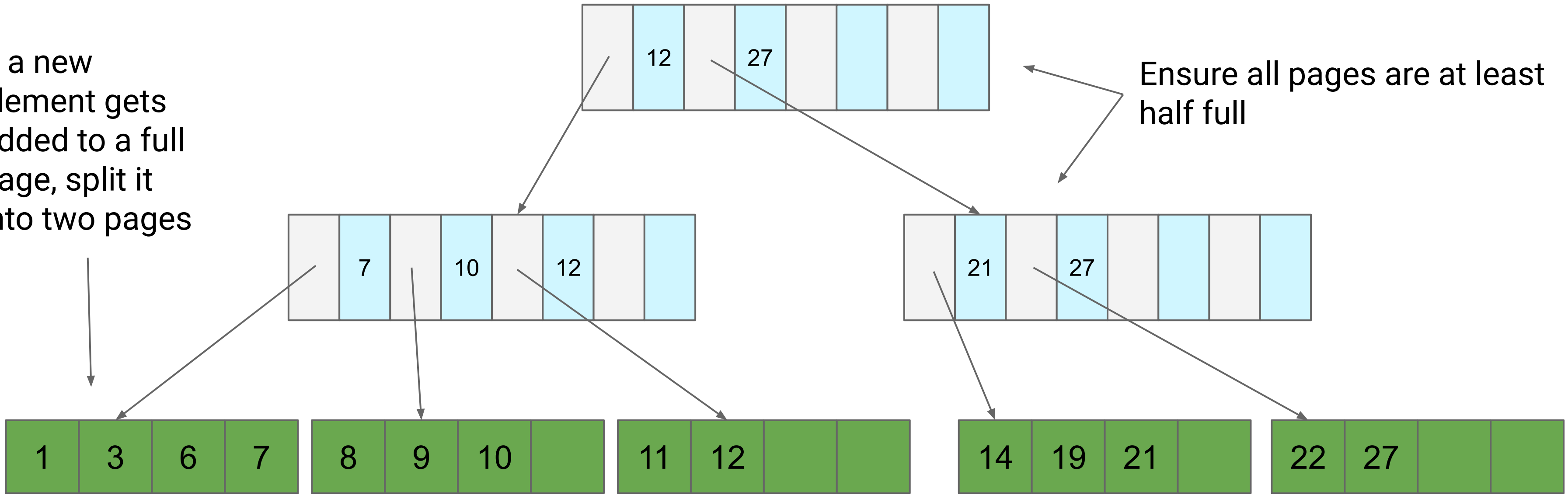
ISAM Index

What if the data changes?

B+ Trees

Keep free space in your pages…but not too much free space

If a new element gets added to a full page, split it into two pages



Ensure all pages are at least half full

Lossy Sets

`LossySet<T>`

`void add(T t)`

- Insert `t` into the set (kind of)

`boolean contains(T t)`

- If `t` is in the set **ALWAYS** return true
- If `t` is not in the set **USUALLY** return false (returning true is OK)

Lossy Set

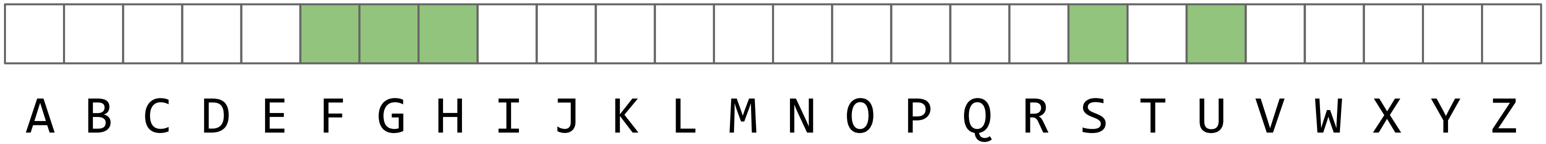
What does this gain for us?

Idea: If apply doesn't always need to be right, we don't need to store everything

Lossy Set Example

```
add("Frankenstein")
add("Get Out")
add("Scream")
add("Hellraiser")
add("Us")
add("Friday the 13th")
```

```
apply("Scream")? TRUE
apply("Saw")? TRUE
apply("The Candyman")? FALSE
apply("Dracula")? FALSE
apply("Friday the 13th")? TRUE
```



Thanks for a great semester!