Lecture 11 - Review of Data Structures

Eric A. Autry

Last time: Complexity Theory

This time: Review of Data Structures

Next time: Recursion and Recurrence Relations

Midterm regrades returned today.

HW 4 due this Thursday the 18th.

Complexity Classifications

While we often report the Big-O runtime for an algorithm, there are actually five ways to classify the complexity.

- $f \in O(g)$ (Big-O, 'f is bounded above by g')
- $f \in o(g)$ (little-o, 'f is strictly bounded above by g')
- $f \in \Omega(g)$ (Big-Omega, 'f is bounded below by g')
- $f \in \omega(g)$ (little-omega, 'f is strictly bounded below by g')
- $f \in \Theta(g)$ (Theta, 'f is bounded both above and below asymptotically by g')

Complexity Classifications

Note that these definitions are all related to each other:

$$
\begin{aligned}
f \in o(g) &\Rightarrow f \in O(g), \\
f \in \omega(g) &\Rightarrow f \in \Omega(g), \\
f \in O(g) &\Rightarrow g \in \Omega(f), \\
f \in o(g) &\Rightarrow g \in \omega(f), \\
f \in \Theta(g) &\Rightarrow f \in O(g) \text{ and } g \in O(f),
\end{aligned}
$$

etc

Complexity Classifications

But what do these complexity classifications actually mean?

How do we interpret them?

When should we use which one?

All of this will depend on what we are using them to report.

Best, Average, and Worst Case

We can report the algorithmic complexity of three different scenarios:

- Best Case: how fast can the algorithm run when we get lucky and use the fewest possible operations?
- Average Case: how fast can the algorithm run on average?
- Worst Case: what is the longest the algorithm could run on an input?

All of these can give insight into the efficiency of an algorithm, though the worst case runtime is the most commonly reported (typically easier to compute and more directly relevant).

Interpreting Complexity

Let's consider several statements that could potentially be made about an algorithm, and what they would mean:

- The best case runtime is $O(f(n))$.
  - The **best** runtime of the algorithm will be at **worst** $f(n)$.
  - i.e., in the best case, it will take less than $f(n)$ time.
- The best case runtime is $\Omega(f(n))$.
  - The **best** runtime of the algorithm will be at **best** $f(n)$.
  - i.e., even in the best case, it must take at least $f(n)$ time.
- The worst case runtime is $O(f(n))$.
  - The **worst** runtime of the algorithm will be at **worst** $f(n)$.
  - i.e., even in the worst case, it will take less than $f(n)$ time.
- The worst case runtime is $\Omega(f(n))$.
  - The **worst** runtime of the algorithm will be at **best** $f(n)$.
  - i.e., in the worst case, it must take at least $f(n)$ time.

Interpreting Complexity

Let's look at two of those statements again:

- The best case runtime is $\Omega(f(n))$.
  - This is the lower bound for how long the algorithm can take.
- The worst case runtime is $O(g(n))$.
  - This is the upper bound for how long the algorithm can take.

If we have these two statements together, it means that if the true runtime of the algorithm is $T(n)$, then

$$k_1 f(n) \leq T(n) \leq k_2 g(n), \quad \text{as} \quad n \to \infty.$$

These are the two most commonly reported runtime statistics.

What about the Constants?

A common question that gets asked is whether we care about the constants.

After all,

$$5n \in O(n) \quad \text{AND} \quad 50n \in O(n).$$

Clearly a runtime of $50n$ will take 10 times longer than a runtime of $5n$, but we report them both as $O(n)$.

We do care about the constants, and the algorithm that gives the runtime of $5n$ will be much better.

The reason we report both algorithms as $O(n)$ is because an algorithm that was $O(n^2)$ would be so much worse...

- For even $n = 100$, the function n^2 will be 100 times worse.
- And we care about performance as the input size becomes very large...

Stack

Data stored in Last In First Out order.

Operations:

- Push: push a new value onto the top in $O(1)$ time
- Pop: pop off the value at the top in $O(1)$ time

Implementations:

- Dynamic Array where $A[0]$ is the bottom of the stack and the length of the stack is stored/updated separately. The array size is doubled when out of space.
- Linked List where values are pushed/popped from the front of the list.

Queue

Data stored in First In First Out order.

Operations:

- Push: push a new value at the back in $O(1)$ time
- Pop: pop a value from the front in $O(1)$ time

Implementations:

- Doubly Linked List with pushing at the front and popping at the end.
- Singly Linked List with pushing at the front and popping at the end, requires a pointer to the last element in the list.
- Dynamic Array with indices tracking the front and the rear that can wrap around, with the array doubling in size when full.

Binary Search Trees (BST)

Keys are stored in a specific sorted order:

- All keys in the right subtree are larger.
- All keys in the left subtree are smaller.

Operations:

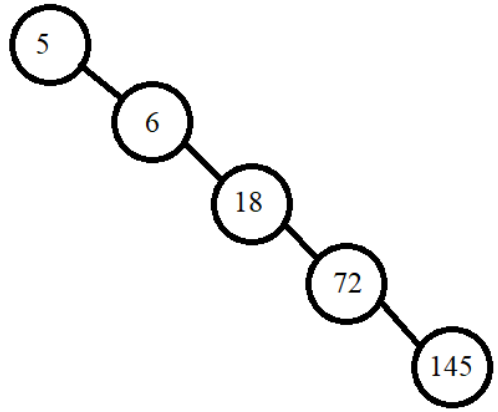
- Insert: insert a new key into the tree in $O(\log n)$ average time and $O(n)$ worst case time.
- Delete: delete a key from the tree in $O(\log n)$ average time and $O(n)$ worst case time.
- Search: search for a key in the tree in $O(\log n)$ average time and $O(n)$ worst case time.

Why $O(\log n)$ average time? What is the depth of the tree?

Why $O(n)$ worst case time? Twigs.

Binary Search Trees (BST)

Why $O(n)$ worst case time? Twigs.



There are many variations of BSTs that try to maintain a balanced tree in an attempt to promise the $O(\log n)$ performance:

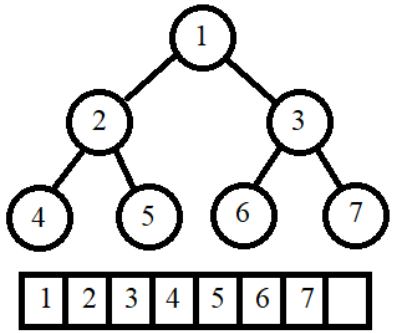
- AVL trees
- B-trees
- Red-black trees
- Splay trees
- etc

Binary Search Trees (BST)

Implementations:

- Recursively stored node structs, where each node keeps its own value, along with pointers to its left child and its right child.
- Dynamic Array where a node's index is determined by where it is in the tree.
  - Layer k of the tree has size at most 2^k (if we call the root layer 0).
  - So if the tree has n layers, will need space:

$$2^0 + 2^1 + 2^2 + \cdots + 2^n = 2^{n+1} - 1.$$



Heap

Special tree-based structure with specified rule:

- Any node's key is always smaller (or larger) than its children's keys.
- We will sometimes call these min-heaps (or max-heaps).
- Note that this rule guarantees that the root will always have the min (or max) key in the heap.

Operations:

- Peek: return the min (or max) value without changing the heap.
- Pop: return the min (or max) value and remove its key from the heap.
- Push: push a new key into the heap.
- Update: update the value of a key within the heap.
- Merge: merge two heaps together.

Heap

There are many different implementations for heaps that all promise different runtimes for the different operations:

- Binary Heap
- Binomial Heap
- Fibonacci Heap
- etc

Important runtime info: for all variations, a pop operation (return the min and remove it) requires $O(\log n)$ time.

Heaps are commonly used to implement priority queues, where the order in the queue depends on an element's key.

If a priority queue always wants to process the minimum key first, then a min-heap can be used to track the minimum key.

Graphs

There are three primary ways that we will store graphs.

- Adjacency List: each vertex has a list of all the vertices that it is adjacent to. Can be stored as an array of linked lists.
- Adjacency Matrix: the origin vertices are the rows and the destination vertices are the columns, with the value a_{ij} in the ith row and jth column corresponding to the weight of the edge from vertex i to vertex j.
- Incidence Matrix: the vertices are the rows and the edges are the columns, where a 1 in entry (i, j) meaning that vertex i is incident on edge j.

Hash Tables

Hash Tables are used to map keys to values.

A hash function is used to compute the index of a 'bin' where a given value can be found.

- A good hash function should avoid collisions and clustering.

The load factor of a hash table is defined as

$$\lambda = \frac{n}{b},$$

where n is the number of keys, and b is the number of bins.

Want a good balance between taking up too much space (small load), and larger access times (large load)

- Ex: Java 10 HashMap load factor default is 0.75

Hash Tables - Implementation

There are two primary methods of implementing a hash table:

- Separate Chaining (typically with linked lists)
  - Each bin can store as many values as it wants.
  - When multiple values all correspond to the same key, all of those values are stored in the same bin.
  - Requires searching through the bin to find a given value.
- Open Addressing (typically with an array)
  - Each bin can hold only a single value.
  - If multiple values all correspond to the same key, a new bin has to be found.
  - When such a collision occurs, consider successive bins until there is an opening, then store the new value there.
  - To find a value, go to its 'original' bin, then search until you either find the value or find a blank space.
  - Good for storing 'small' value due to better locality for caching.
  - Requires a very good hash function and a low load factor.