x-Fast and y-Fast Tries
Outline for Today

- **Data Structures on Integers**
  - How can we speed up operations that work on integer data?

- **x-Fast Tries**
  - Bit manipulation meets tries and hashing.

- **y-Fast Tries**
  - Combining RMQ, strings, balanced trees, amortization, and randomization!
Working with Integers

- Many practical problems involve working specifically with integer values.
  - **CPU Scheduling**: Each thread has some associated integer priority, and we need to maintain those priorities in sorted order.
  - **Network Routing**: Each computer has an associated IP address, and we need to figure out which connections are active.
  - **ID Management**: We need to store social security numbers, zip codes, phone numbers, credit card numbers, etc. and perform basic lookups and range searches on them.
- We’ve seen many general-purpose data structures for keeping things in order and looking things up.
- **Question**: Can we improve those data structures if we know in advance that we’re working with integer data?
Working with Integers

• Integers are interesting objects to work with:
  • Their values can directly be used as indices in lookup tables.
  • They can be treated as strings of bits, so we can use techniques from string processing.
  • They fit into machine words, so we can process the bits in parallel with individual word operations.

• The data structures we’ll explore over the next few lectures will give you a sense of what sorts of techniques are possible with integer data.
An Auxiliary Motive

- Integer data structures are also a great place to see just how much you’ve learned over the quarter!
- Today’s data structures cover every single unit from the quarter (RMQ, strings, balanced trees, amortization, and randomization).
- I hope this gives you a chance to pause and reflect on just how far you’ve come!
The Setup
Our Machine Model

- We will assume we’re working on a machine where memory is segmented into \( w \)-bit words.
- We’ll assume our integers are drawn from some set \([U]\), where \( \lg U = O(w) \).
  - In other words, we assume our integers fit into a constant number of machine words.
- We’ll assume that the C integer operators work in constant time, and will not assume we have access to operators beyond them.

\[ + - * / \% \ll \gg \& | \wedge = \leq \]
Ordered Dictionaries
Ordered Dictionaries

- An *ordered dictionary* maintains a set $S$ drawn from an ordered universe $\mathcal{U}$ and supports these operations:
  - `lookup(x)`, which returns whether $x \in S$;
  - `insert(x)`, which adds $x$ to $S$;
  - `delete(x)`, which removes $x$ from $S$;
  - `max()` / `min()`, which return the maximum or minimum element of $S$;
  - `successor(x)`, which returns the smallest element of $S$ greater than $x$; and
  - `predecessor(x)`, which returns the largest element of $S$ smaller than $x$.

- For context:
  
  Ordered Dictionary : BST :: Queue : Linked List
Ordered Dictionaries

- Balanced BSTs support all ordered dictionary operations in time $O(\log n)$ each.
- Hash tables support insertion, lookups, and deletion in expected time $O(1)$, but require time $O(n)$ for max, min, successor, and predecessor.

**Question:** Can we improve upon these bounds if we know that we’re working with integers drawn from $[U]$?
A Start: *Bitwise Tries*
Tries Revisited

- **Recall:** A trie is a simple data structure for storing strings.

- Integers can be thought of as strings of bits.

- **Idea:** Store integers in a *bitwise trie*. 
Finding Successors

- To compute \textit{successor}(x), do the following:
  - Search for \( x \).
  - If \( x \) is a leaf node, its successor is the next leaf.
  - If you don't find \( x \), back up until you find a node with a 1 child not already followed, follow the 1, then take the cheapest path down.
Bitwise Trie Efficiency

- All operations on bitwise tries take time proportional to the number of bits in each number.
- Runtime for each operation: $O(\log U)$.
  - This is probably worse than $O(\log n)$.
- For each number stored, we need to store $\Theta(\log U)$ internal nodes.
- Space usage: $O(n \log U)$.
  - This is probably worse than a BST.
- *Can we do better?*
Speeding up Successors

• There are two independent pieces that contribute to the $O(\log U)$ runtime:
  • Need to walk down the trie following the bits of $x$, and there are $\Theta(\log U)$ of those.
  • From there, need to back up to a branching node where we can find the successor.

• Can we speed up those operations? Or at least work around them?
**Observation:** A lookup for $x$ in this trie terminates at the node corresponding to the longest prefix of $x$.

**Question:** Do we actually have to walk the trie to find this node?
Claim 1: If a node $v$ corresponds to a prefix of $x$, all of $v$'s ancestors correspond to prefixes of $x$. 
Claim 2: If a node $v$ does not correspond to a prefix of $x$, none of $v$'s descendants correspond to prefixes of $x$. 
Claim 3: The deepest node corresponding to a prefix of $x$ can be found by doing a binary search over the layers of the trie.
One Speedup

- **Goal:** Encode the trie so that we can do a binary search over its layers.

- **One Solution:** Store an array of cuckoo hash tables, one per layer of the trie, that holds all the nodes in that layer.

- Can now query, in worst-case time $O(1)$, whether a node's prefix is present on a given layer.

- There are $O(\log U)$ layers in the trie.

- Binary search will take worst-case time $O(\log \log U)$.

- **Nice side-effect:** Queries are now worst-case $O(1)$, since we can just check the hash table at the bottom layer.
Performing the Binary Search

• This binary search assumes that, given a number $x$ and a length $k$, we can extract the first $k$ bits of $x$ in time $O(1)$.

• Fortunately, we can do this!

```c
uint64_t x = /* … */;
uint64_t mask = -(uint64_t(1) << (64 - k));
uint64_t prefix = x & mask;
```

There’s an edge case to handle here for $k = 0$, but that’s easily special-cased. Let me know if there’s a way to avoid this!
Finding Successors

• We can now find the node where the successor search would initially arrive.

• At this point, we’d normally back up until we find a branching node where we can follow a 1 child pointer, then descend from there to the leaves.

• This will take time $O(\log U)$.

• Can we do better?
Finding Successors

- **Claim:** If the binary search terminates at a node $v$, that node must have at most one child.
- If it doesn't, it has both a 0 child and a 1 child, so there's a longer prefix that can be matched.
- **Idea:** Steal the missing pointers and use them to speed up successor and predecessor searches.
Threaded Binary Tries

- A **threaded binary trie** is a binary tree where
  - each missing 0 pointer points to the inorder predecessor of the node and
  - each missing 1 points to the inorder successor of the node.
  - Notice that the leaves end up in a doubly-linked list.
**x-Fast Tries**

- An **x-Fast Trie** is a threaded binary trie with a cuckoo hash table at each level that stores the nodes at that level.
- Can do lookups in time $O(1)$. 

x-Fast Tries

- **Claim:** Can determine \( \text{successor}(x) \) in time \( O(\log \log U) \).

- Start by binary searching for the longest prefix of \( x \).

- If that node has a missing 1 pointer, it points directly to the successor.

- Otherwise, it has a missing 0 pointer.

- If that pointer is null, return the minimum value (we can cache this.)

- Otherwise, follow it to a leaf, then follow the leaf’s 1 pointer.
Based on what we've seen:

- *lookup* takes worst-case time $O(1)$.
- *successor* and *predecessor* queries take worst-case time $O(\log \log U)$.
- *min* and *max* can be done in time $O(1)$, assuming we cache those values.

- How efficiently can we support *insert* and *delete*?
• If we $\text{insert}(x)$, we need to
  • add some new nodes to the trie;
  • wire $x$ into the doubly-linked list of leaves; and
  • update the thread pointers to include $x$.
• Worst-case will be $\Omega(\log U)$ due to the first and third steps.
x-Fast Tries

- Here is an (amortized, expected) $O(\log U)$ time algorithm for $\text{insert}(x)$:
  - Find $\text{successor}(x)$.
  - Add $x$ to the trie.
  - Using the successor from before, wire $x$ into the linked list.
  - Walk up from $x$, its successor, and its predecessor and update threads.
x-Fast Tries

- Here is an (amortized, expected) $O(\log U)$ time algorithm for $\text{insert}(x)$:
  - Find $\text{successor}(x)$.
  - Add $x$ to the trie.
  - Using the successor from before, wire $x$ into the linked list.
  - Walk up from $x$, its successor, and its predecessor and update threads.
x-Fast Tries

- Here is an (amortized, expected) $O(\log U)$ time algorithm for $\text{insert}(x)$:
  - Find $\text{successor}(x)$.
  - Add $x$ to the trie.
  - Using the successor from before, wire $x$ into the linked list.
  - Walk up from $x$, its successor, and its predecessor and update threads.
Deletion

- To delete \( x \), we need to
  - Remove \( x \) from the trie.
  - Splice \( x \) out of its linked list.
  - Update thread pointers from \( x \)'s former predecessor and successor.
- Runs in expected, amortized time \( O(\log U) \).
- Full details are left as a proverbial Exercise to the Reader.
Space Usage

• How much space is required in an $x$-fast trie?
• Each leaf node contributes at most $O(\log U)$ nodes in the trie.
• Total space usage for hash tables is proportional to total number of trie nodes.
• Total space: $O(n \log U)$. 
Where We Stand

- Right now, we have a reasonably fast data structure for storing a sorted set of integers.
- If we have a static set of integers that we want to make lots of queries on, this is pretty good as-is!
- As you’ll see, though, we can make this even better with some kitchen sink techniques. 😊

x-Fast Trie:
- **lookup**: $O(1)$
- **insert**: $O(\log U)^*$
- **delete**: $O(\log U)^*$
- **max**: $O(1)$
- **succ**: $O(\log \log U)$
- **is-empty**: $O(1)$
- Space: $O(n \log U)$

* Expected, amortized
Time-Out for Announcements!
Midterm Logistics

● Our midterm will be held next Tuesday from 7:00PM – 10:00PM in **Hewlett 200**.

● Exam is closed-book, closed-computer, and limited-note. You can bring a double-sided 8.5” × 11” sheet of notes with you to the exam.

● Topic coverage is material from PS1 – PS5. Topics from this week won’t be tested, but are an excellent review of the concepts.

● We've released a set of practice problems to help you prepare for the exam. They're up on the course website.

● Can't make the exam time? Have OAE accommodations? Let us know ASAP so that we can set up an alternate time.
Final Project Presentations

- Final project presentations will run from *Monday, June 4* to *Thursday, June 7*.
- Use this link to sign up for a time slot: [http://www.slottr.com/cs166-2018](http://www.slottr.com/cs166-2018)
- You can view the available time slots starting today. The form will be open from *noon on Thursday, May 24* until noon on Thursday, May 31. It's first-come, first-served.
- Presentations will be 15-20 minutes, plus five minutes for questions. Please arrive five minutes early to get set up.
- Presentations are open to the public, so feel free to stop by any of the presentations you're interested in.
Back to CS166!
y-Fast Tries
Where We Stand

- Right now, we have a reasonably fast data structure for storing a sorted set of integers.
- To make this really shine, we need to improve the highlighted costs.

\[
\begin{array}{|c|c|}
\hline
\text{Operation} & \text{Time Complexity} \\
\hline
\text{lookup} & \mathcal{O}(1) \\
\text{insert} & \mathcal{O}(\log U) \\
\text{delete} & \mathcal{O}(\log U) \\
\text{max} & \mathcal{O}(1) \\
\text{succ} & \mathcal{O}(\log \log U) \\
\text{is-empty} & \mathcal{O}(1) \\
\hline
\end{array}
\]

- Space: \( \Theta(n \log U) \)

* Expected, amortized
We’re essentially at a spot where we need to shave off a log factor from a couple of operations.

**Question**: What techniques have we developed so far to do this?

\[ \text{x-Fast Trie:} \]
- **lookup**: \( O(1) \)
- **insert**: \( O(\log U) \)*
- **delete**: \( O(\log U) \)*
- **max**: \( O(1) \)
- **succ**: \( O(\log \log U) \)
- **is-empty**: \( O(1) \)
- Space: \( \Theta(n \log U) \)

* Expected, amortized
Two-Level Structures

- Think back to the hybrid approach we used for solving RMQ.
- It consisted of a two-tiered structure:
  - A bunch of small, lower-level structures that each solve the problem in small cases.
  - A single, larger, top-level structure that helps aggregate those solutions together.
Two-Level Structures

- One of the fastest RMQ hybrids in practice is the $\langle O(n), O(\log n) \rangle$ hybrid structure built with blocks of size $\Theta(\log n)$ where
  - the summary structure is a $\langle O(n \log n), O(1) \rangle$ sparse table, and
  - the block-level structures are $\langle O(1), O(n) \rangle$ no-preprocessing RMQ structures.
- By breaking the input apart into blocks of size $\Theta(\log n)$
  - the summary structure only takes time $O(n)$ to build, and
  - the linear terms in the blocks become $O(\log n)$ terms.
The Idea

- Build a two-level ordered dictionary out of existing ordered dictionaries.
- Split the keys apart into logarithmic-sized blocks.
- Build ordered dictionaries for each of the block-level dictionaries.
- Build a summary dictionary to aggregate the blocks together.
The $y$-Fast Trie
The Setup

- For now, assume all keys are given to us in advance, in sorted order.
- Split the keys apart into blocks of size $\Theta(\log U)$ and store them in balanced BSTs.

1 3 7 27 31 41 59 79 90 103 106 107 109 110 161
The Setup

- For now, assume all keys are given to us in advance, in sorted order.
- Split the keys apart into blocks of size $\Theta(\log U)$ and store them in balanced BSTs.
- Create a summary $x$-fast trie that stores the maximum key from each block but the last.
Performing a Lookup

- Suppose we want to perform lookup(90).
- **Idea:** figure out which block 90 would belong to, then search within the BST in that block.
- Cost: $O(\log \log U)$.

Ask for **successor**(89) up here in time $O(\log \log U)$.

Search this BST in time $O(\log \log U)$. 
Performing a Lookup

- Suppose we want to perform \textit{lookup}(110).
- \textbf{Idea:} figure out which block 109 would belong to, then search within the BST in that block.
- Cost: $O(\log \log U)$.

\begin{itemize}
  \item Search this BST in time $O(\log \log U)$.
  \item Ask for \textit{successor}(109) in time $O(\log \log U)$.
  \item (Oops, doesn't exist!)
\end{itemize}
Successor Queries

- How might we perform *successor* queries?
- Here’s how we’d determine *successor*(59).

Cost: $O(\log \log U)$.

Ask for *successor*(58) up here in time $O(\log \log U)$.

Find successor in time $O(\log \log U)$. 
Successor Queries

• How might we perform successor queries?
• Here’s how we’d determine successor(107).
• Cost: $O(\log \log U)$.

![x-Fast Trie Diagram](image)

Ask for successor(106) up here in time $O(\log \log U)$.

Find successor in time $O(\log \log U)$. (Oops, doesn’t exist!)

Find min in time $O(\log \log U)$. 
Making Edits

- With a major caveat, insertions follow the same procedure as before.
- Here’s how we’d **insert** (6) into this BST in time $O(\log \log U)$.

$\text{Ask for } \text{successor}(5) \text{ in time } O(\log \log U)$. 

**x-Fast Trie**

| 7 | 41 | 90 | 107 |

| 3 |

| 1 | 7 |

| 27 | 41 |

| 59 | 90 |

| 103 | 107 |

| 110 |

| 109 | 161 |
The Problem

- If our trees get too big, we may lose our $O(\log \log U)$ time bound.
- **Idea:** Require each tree to have at most $2 \cdot \lg U$ elements. If it gets too big, split it and update the $x$-fast trie.
The Problem

- If our trees get too big, we may lose our $O(\log \log U)$ time bound.
- **Idea:** Require each tree to have at most $2 \cdot \lg U$ elements. If it gets too big, split it and update the $x$-fast trie.
Analyzing an Insertion

• If we perform an `insert` and don’t end up doing a resize, the cost is $O(\log \log U)$.

• If we perform an `insert` and do have to do a resize, the work done is
  • $O(\log \log U)$ to `split` the binary search tree, and
  • $O(\log U)$ to insert into the $x$-fast trie.

• Total work: $O(\log U)$. 
An Amortized Analysis

- Whenever we do an insertion, place a credit on the newly-inserted element.
  - Cost of a “light” insert still $O(\log \log U)$.
- If we have to split a tree, the tree size was above $2 \log U$, so there must be $\log U$ credits on it (one for each element above $\log U$).
- The \textit{amortized} cost of a “heavy” insert is then
  \[O(\log \log U) + O(\log U) - \Theta(\log U) = O(\log \log U).\]

- Cost of a regular insert, plus the tree split.
- Cost of adding to the $x$-fast trie.
- Credits spent.
A Nice Side-Effect

- We can now abandon our assumption that we’re given all the keys in sorted order in advance.
- Each insertion takes amortized time $O(\log \log U)$, so we can build the structure up from scratch!

This is an (expected) $O(n \log \log U)$-time sorting algorithm!
Making Edits

• With a major caveat, deletions follow the same procedure as insertions.
• Here’s how we’d delete(7).

```

delete from this BST in time $O(\log \log U)$
```

```
x-Fast Trie

| 7 | 41 | 90 | 107 |
```

```
Ask for successor(6) in time $O(\log \log U)$.
```
Making Edits

- Our x-fast trie still holds 7, even though 7 is no longer present.
- That’s not a problem – those keys just serve as “routing information” to tell us which BSTs to look at.
- **Intuition:** The x-fast trie keys act as partitions between BSTs. They don’t need to actually be present in our data structure.
Shrinking our Structure

- What happens if we remove all the elements from our structure without touching the x-fast trie?
- Each operation still takes time $O(\log \log U)$.
- But now our space usage depends on the maximum size we reached, not the current size!
Achieving a Balance

- If each tree has $\Theta(\log U)$ elements in it, then our space usage is
  - $\Theta(n)$ for all the trees, plus
  - $\Theta((n / \log U) \log U) = \Theta(n)$ for the $x$-fast trie,
- This uses $\Theta(n)$ total memory.
Achieving a Balance

- **Invariant**: Require each tree to have between $\frac{1}{2}\log_2 U$ and $2\log_2 U$ elements.

- If a tree gets too small, either
  - borrow lots of elements from a neighbor and update the x-fast trie, or
  - merge with a neighbor and update the x-fast trie.
What We’ve Seen

- Here’s the final scorecard for the y-fast trie.

- Assuming \( n = \omega(\log U) \), which it probably is, this is strictly better than a binary search tree.

- And it gives rise to an \( O(n \log \log U) \)-expected-time sorting algorithm!

The y-Fast Trie:

- \textbf{lookup}: \( O(\log \log U) \)
- \textbf{insert}: \( O(\log \log U) \)*
- \textbf{delete}: \( O(\log \log U) \)*
- \textbf{max}: \( O(\log \log U) \)
- \textbf{succ}: \( O(\log \log U) \)
- \textbf{is-empty}: \( O(1) \)
- Space: \( \Theta(n) \)

* Expected, amortized.
What We Needed

- An $x$-fast trie requires *tries* and *cuckoo hashing*.
- The $y$-fast trie requires amortized analysis and *split/join* on *balanced BSTs*.
- $y$-fast tries also use the “blocking” technique from *RMQ* we used to shave off log factors.
What’s Missing

• There’s still a little gap between where BSTs dominate and where $y$-fast tries take over.
  • Specifically, what if $n = O(\log U)$?

• Our solution still involves randomness.
  • We need that in the cuckoo hash tables at each level.

• **Question:** Can we build a solution with neither of these weaknesses?
Next Time

• **Word-Level Parallelism**
  • Parallel processing via addition, subtraction, and the like.

• **Sardine Trees**
  • A fast ordered dictionary for truly tiny trees.