Unit Overview
Tables & Priority Queues

Major Topics
✓ ▪ Introduction to Tables
✓ ▪ Priority Queues
✓ ▪ Binary Heap algorithms
✓ ▪ Heaps & Priority Queues in the C++ STL
✓ ▪ 2-3 Trees
✓ ▪ Other balanced search trees
(part) ▪ Hash Tables
  ▪ Prefix Trees
  ▪ Tables in various languages
Review
Introduction to Tables

<table>
<thead>
<tr>
<th></th>
<th>Sorted Array</th>
<th>Unsorted Array</th>
<th>Sorted Linked List</th>
<th>Unsorted Linked List</th>
<th>Binary Search Tree</th>
<th>Balanced (how?) Binary Search Tree</th>
</tr>
</thead>
<tbody>
<tr>
<td>Retrieve</td>
<td>Logarithmic</td>
<td>Linear</td>
<td>Linear</td>
<td>Linear</td>
<td>Linear</td>
<td>Logarithmic</td>
</tr>
<tr>
<td>Insert</td>
<td>Linear</td>
<td>Constant (?)</td>
<td>Linear</td>
<td>Constant</td>
<td>Linear</td>
<td>Logarithmic</td>
</tr>
<tr>
<td>Delete</td>
<td>Linear</td>
<td>Linear</td>
<td>Linear</td>
<td>Linear</td>
<td>Linear</td>
<td>Logarithmic</td>
</tr>
</tbody>
</table>

Idea #1: Restricted Table
- Perhaps we can do better if we do not implement a Table in its full generality.

Idea #2: Keep a Tree Balanced
- Balanced Binary Search Trees look good, but how to keep them balanced efficiently?

Idea #3: “Magic Functions”
- Use an unsorted array of key-data pairs. Allow array items to be marked as “empty”.
- Have a “magic function” that tells the index of an item.
- Retrieve/insert/delete in constant time? (Actually no, but this is still a worthwhile idea.)

We will look at what results from these ideas:
- From idea #1: Priority Queues
- From idea #2: Balanced search trees (2-3 Trees, Red-Black Trees, B-Trees, etc.)
- From idea #3: Hash Tables
Overview of Advanced Table Implementations

We will cover the following advanced Table implementations.

- Balanced Search Trees
  - Binary Search Trees are hard to keep balanced, so to make things easier we allow more than 2 children:
    - 2-3 Tree
      - Up to 3 children
    - 2-3-4 Tree
      - Up to 4 children
    - Red-Black Tree
      - Binary-tree representation of a 2-3-4 tree
  - Or back up and try a balanced Binary Tree again:
    - AVL Tree
- Alternatively, forget about trees entirely:
  - Hash Tables
- Finally, “the Radix Sort of Table implementations”:
  - Prefix Tree
A **Hash Table** is a Table implementation that uses a **hash function** for key-based look-up.

- A Hash Table is generally implemented as an array. The index used is the output of the hash function.

Needed:

- Hash function.
- **Collision resolution** method.
  - **Collision**: hash function gives same output for different keys.
A hash function **must**:
- Take a valid key and return an integer.
- Be **deterministic**.
  - Its value depends only on its input (the key). Using the same input multiple times results in the same output each time.

A **good** hash function:
- Can be computed quickly.
- Spreads out its results evenly over the possible output values.
  - To help spread out the results, some implementations give the Hash Table a **prime** number of locations.
- Turns patterns in its input into random-looking output.

Each key type has its own hash function.
- For client-defined key types, a hash function must be provided by the client.
- Can put different key types, each with its own hash function, in the same Hash Table.
- Hash Table sends the output of the provided hash function through a secondary function ("%"?) to make the output a valid index.
Recall that a **collision** is when the hash function produces the same output for different keys.

- We cannot guarantee that collisions will be rare.

**How collisions are resolved is the primary design decision involved in a Hash Table.**

- Different collision-resolution methods result in different Hash-Table implementations.

Two categories of collision-resolution methods:

- **Open Addressing**
  - The Hash Table is essentially an array of data items.
  - If we get a collision, we look for another spot.
- **“Buckets”**
  - Each location in the Hash Table is a data structure capable of storing multiple data items.
  - In this case, a location in the Hash Table is called a **bucket**.
In **open addressing**:  
- The Hash Table is essentially an array of data items.  
- Each location can be marked as “empty”.  

When inserting or retrieving (including the retrieve done as part of a delete), we look at a sequence of locations.  
- The first is the location given by the hash function.  
- We continue looking until we find the given key or we are sure it is not present.  
- Each time we view a location, we are doing a **probe**. The entire sequence of locations to view is called the **probe sequence**.
The simplest probe sequence is the one in which we look at location \( t \), then \( t+1 \), then \( t+2 \), etc. This is **linear probing**.

- Linear probing tends to form **clusters**, which slow things down.

To avoid clusters, we can use **quadratic probing**, which has the probe sequence \( t, t+1^2, t+2^2, t+3^2, \) etc.
Fancier techniques involve using a secondary hash function when a collision occurs.

- These generally go under the name of **double hashing**.
- For example, do a variation of linear probing with a step size other than 1. The step size is given by the second hash function.
- Or, just use the second hash function to give a second location, after which one of the simpler probe sequences is used.
Open addressing leads to a problem: How to be sure a key is not present?

In the Hash Table above, retrieve 3 (not present). Use linear probing.
- Begin at the arrow, look until we find an “empty”, return NOT FOUND.

Now insert 3 and then delete 1.

Again retrieve 3. Using the above strategy, we return NOT FOUND. 😞
Solution: Allow **DELETED** marks. Stop when we find given key or EMPTY.

But now, the array can fill up with “deleted” marks, which slows it down.
Another collision-resolution idea is to make the Hash Table an array of data structures.
  - Each structure can hold multiple data items.
  - We call the locations in the Hash Table “buckets”.

Very common: Make each bucket a Linked List. This is called separate chaining.
  - Why do we not need a Doubly Linked List?

Why not make each bucket a Red-Black Tree?
Idea: use an array-based Linked List for each bucket.

- The Hash Table can be a big array divided into two sections:
  - One section for the heads of the buckets.
    - This section is indexed using the output of the hash function.
  - One section for the rest of the nodes in the buckets.
- Each node is an array element.
- Pointers are replaced by array indices.
- Efficiency is much the same as for a pointer-based Linked List.
  - But memory-management overhead is reduced.
Hash Tables
Table-Remake

Sometimes it is necessary to remake the Hash Table.
  • The Hash Table may fill up, requiring a larger array.
  • All implementations have performance degradation as the number of data items rises.

In these cases, we need to do a reallocate-and-copy, as we did with smart arrays.

Here, however, the copy can be very time-consuming.
  • We need to traverse the entire table, possibly including empty locations.
  • We need to call the hash function for every key present.

This is one of the downsides of Hash Tables.
Hash Tables
Efficiency — Introduction

A **perfect hash function** (one without collisions) results in insert, delete, and retrieve operations that are $O(1)$.

- In practice, we cannot guarantee this, if we allow insert & delete operations.

In the **worst case**, all items get the same hashed value, and so collisions happen nearly all the time.

- Thus, retrieve is linear time, for most implementations.
- *But what if our buckets are Red-Black Trees?*

However, we generally use a Hash Table when we are interested in **average-case** performance.

The average performance of a Hash Table can be analyzed based on the **load factor**.

- The *load factor*, denoted by $\alpha$, is:
  
  $(\# \text{ of items present}) \div (\# \text{ of locations in table})$

- We generally want $\alpha$ to be small. In the following slides, we will assume $\alpha$ is significantly less than 1 (less than 2/3, maybe?).

- We will also assume, for now, that no Table-remake is required.
Hash Tables
Efficiency — Separate Chaining

For example, consider separate chaining.

- **Worst Case**
  - Insert is constant time, assuming we do not search.
    - We can avoid a search, if we allow duplicate keys.
  - Retrieve and delete require a search: linear time.
  - Similarly, if we do not allow duplicate keys, then insert requires a search, and so is linear time.

- **Average Case**
  - The average number of items in a bucket is $\alpha$.
  - Thus, the average number of comparisons required for a search resulting in NOT FOUND is $\alpha$.
  - The average number of comparisons required for a search resulting in FOUND is approximately $1 + \alpha/2$.
  - This applies to operations requiring a search: retrieve and delete certainly, insert maybe. Insert without search is constant time.
Hash Tables
Efficiency — Open Addressing

With open addressing, retrieve, insert, and delete all require a search.

Worst Case
• Linear time.

Average Case
• For linear probing:
  ▪ NOT FOUND: \( (1/2)[1+1/(1-\alpha)]^2 \).
  ▪ FOUND: \( (1/2)[1+1/(1-\alpha)] \).
• For quadratic probing:
  ▪ NOT FOUND: \( 1/(1-\alpha) \).
  ▪ FOUND: \(-\ln(1-\alpha)/\alpha\).
• Again:
  ▪ We assume \( \alpha \) is significantly less than 1, and that the Table-remake operation is not done.
  ▪ The efficiency of insert, delete, and retrieve is essentially the same in all cases.
Hash Tables
Efficiency — Traverse

Hash Table traverse can be slow, because of the empty locations.

- Assume:
  - Either open addressing is used, or else buckets are implemented using structures that can be traversed in linear time.
  - We do not want a sorted traverse.
- Then traverse is $O(n + b)$, where $n$ is the number of items in the Hash Table, and $b$ is the number of locations (buckets?).

A speed up: use an auxiliary Doubly Linked List containing all stored key-data pairs.

- Each key-data pair gets two pointers (previous node, next node).
- Table insert & delete modify the Linked List.
- Table traverse uses the Linked List. Result: traverse is $O(n)$. 

![Diagram of Hash Table and Linked List](attachment:hash_table_linked_list.png)
Hash Tables
Efficiency — Issues

The Table-remake operation has a similar effect on Hash-Table efficiency to that of reallocate-and-copy on a smart array.

• Constant time becomes amortized constant time.

All reasonable implementations of a Hash Table have **average-case** performance of constant time for retrieve and delete, and also for insert, if no Table-remake is required.

• For the insert operation, this becomes an average case of amortized constant time, if Table-remake operations are done intelligently.

In common Hash Table implementations, **worst-case** performance is linear time for retrieve and delete, and also for insert, if duplicate keys are not allowed.

An important issue is whether a **malicious user** can force worst-case performance.

• A well-chosen hash function makes this difficult.
• The design of such a function is beyond the scope of this class, but information and implementations are not hard to find.
Hash Tables
Efficiency — Comparison

<table>
<thead>
<tr>
<th>Idea #1</th>
<th>Idea #2</th>
<th>Idea #3</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Priority Queue using Heap</strong></td>
<td><strong>Red-Black Tree</strong></td>
<td><strong>Hash Table:</strong> average case</td>
</tr>
<tr>
<td>Retrieve</td>
<td>Constant*</td>
<td>Logarithmic</td>
</tr>
<tr>
<td>Insert</td>
<td>(Amortized)** logarthmic</td>
<td>Logarithmic</td>
</tr>
<tr>
<td>Delete</td>
<td>Logarithmic*</td>
<td>Logarithmic</td>
</tr>
</tbody>
</table>

*Priority Queue retrieve & delete are not Table operations in their full generality. Only the item with the highest priority can be retrieved/deleted.

**This is logarithmic if (1) the PQ does not manage its own memory, or (2) enough memory is preallocated. Otherwise, occasional linear-time reallocate-and-copy may be required. Time per-operation, averaged over many consecutive operations, will be logarithmic. Thus, “amortized logarithmic”.

***So Hash Table insert is only “constant time” in a double-average sense: averaged over many consecutive operations, for average data.

****For a typical implementation, using either open addressing or separate chaining, with duplicate keys not allowed.
Hash Tables
Efficiency — Conclusion

We have another example of average-case vs. worst-case efficiency trade-off.

- One that we saw was Quicksort vs. $O(n \log n)$ sorts. But we do not need to worry about that any more.
- However, Hash Tables vs. balanced search trees is still an issue.

Hash Tables have very good performance for “typical” situations.

- Its occasional drawbacks can be serious.

When using a Hash Table, do so intelligently.
Prefix Trees
Background

Consider a list of words.

- In practice, our list might be much longer.
- Alphabetically order the words. Each is likely to have many letters in common with its predecessor.
- That is, consecutive words tend to have a prefix in common.

One easy way to take advantage of this is to store each word as a number followed by letters.

- This method is very suitable for use in a text file that is loaded all at once.
- But it does not support fast look-up by key (word).

A method more suited for in-memory use is a Prefix Tree.

- Also called “Trie” for “reTRIEval”.
  - You’re supposed to say “TREE”. 😊 I’ve heard “TRY”. 😊
  - Ick.

\[
\begin{array}{c}
dig \\
dog \\
dot \\
dote \\
doting \\
\end{array}
\quad
\begin{array}{c}
dig \\
log \\
2t \\
3e \\
3ing \\
0eggs \\
\end{array}
\]

Not a Prefix Tree!
A **Prefix Tree** (or **Trie**) is a Table implementation in which the keys are strings.

- We use “string” in a general sense, as in our discussion of Radix Sort.
  - A nonnegative integer is a string of digits.
- The quintessential key type is **words**, as in the previous slide.
- A Prefix Tree is space-efficient when keys tend to share prefixes.

A Prefix Tree is a kind of tree.

- Each node can have one child for each possible character.
- Each node also contains a Boolean value, indicating whether it represents a stored key.
  - Duplicate keys are not allowed.
- Lastly, each node can hold the data associated with a key.
Prefix Trees
Definition [2/2]

In a Prefix Tree for storing lower-case English words, each node has:

- 26 child pointers (one for each letter).
- A Boolean value
- A spot for the associated data.

The keys in the Prefix Tree to the right are those from our word list: **dig, dog, dot, dote, doting, eggs.**

- Rather than draw 26 pointers for each node, I have labeled each pointer with the appropriate letter.
- A node with a black circle is one that represents a word in the list.
Prefix Trees
Implementation

How would we implement a Prefix Tree node?

- Example:

```c
struct PTNode {
    (PTNode *) ptrs_[26]; // a .. z ptrs; NULL if none
    bool isWord_;         // true if a word ends here
    DataType data_;      
};
```

- Another possibility:

```c
struct PTNode {
    std::map<char, PTNode *> ptrs_;  // An STL Table implementation (think “Red-Black Tree”)
    bool isWord_;                     
    DataType data_;                  
};
```

An RAII class would be good to have here. See Boost’s `shared_ptr`.
Prefix Trees
Any Good?

Efficiency

- For a Prefix Tree, Table retrieve, insert, and delete all take a number of steps proportional to the length of the key.
- If word length is considered fixed, then all are constant time.
- However, word length is logarithmic in the number of possible words.
  - A hidden logarithm, just like Radix Sort.

A Prefix Tree is a good basis for a Table implementation, when keys are short-ish sequences from a not-too-huge alphabet.

- Words in a dictionary, ZIP codes, etc.
- Just like Radix Sort.

A Prefix Tree is easy to implement.

The idea behind Prefix Trees is also used in other data structures.