Unit Overview
Tables & Priority Queues

Major Topics:
✓ • Introduction to Tables
✓ • Priority Queues
✓ • Heap algorithms
✓ • Heaps and Priority Queues in practice
✓ • 2-3 Trees
✓ • Other balanced search trees
✓ • Hash Tables
  • Prefix Trees
  • Tables in practice
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 do we keep them balanced efficiently?

Idea #3: “Magic Functions”
- Use an unsorted array and allow 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 into what results from these ideas:
- Priority Queues
- Balanced search trees (2-3 Trees, Red-Black Trees, etc.)
- Hash Tables
Advanced Table Implementations
Overview

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**
- Then, “the Radix Sort of Table implementations”:
  - **Prefix Tree**
A **Hash Table** is a Table ADT 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:**

- A hash function.
- A method for resolving **collisions**.
  - *Collision*: hash function gives the same output for different keys.
  - This is the most important design decision for a Hash Table.
A hash function ...

- Takes a valid key and returns an integer.
- **Must** be **deterministic**.
- **Should** be fast, “spread out” results, produce difficult-to-predict results given patterned input.
  - The last property is the hard one.

For non-built-in value types, the hash function must be provided by the client.

To spread out the results, it can help for the Hash Table to have a **prime** number of locations.
Collision Resolution Methods — Type 1: **Open Addressing**

- Table is an array. Each location holds a data item, “empty”, or “deleted”.
- Search in a sequence of locations (the **probe sequence**), beginning at the location given by the hashed key.
  - **Linear probing**: $t, t+1, t+2$, etc.
    - Tends to produce **clusters**.
  - **Quadratic probing**: $t, t+1^2, t+2^2$, etc.
  - **Double hashing**: Use another hash function to help determine the probe sequence.
Collision Resolution Methods — Type 2: **Buckets**

- Table is an array of data structures, each of which can hold multiple items.
  - Linked lists are common.
- Array locations are **buckets**.

![Diagram of buckets with linked lists]
Review
Hash Tables — Efficiency, etc.

Efficiency

- The load factor (often “$\alpha$”) of a Hash Table is the number of items divided by the number of locations (buckets).
  - Thus, $\alpha$ is the average number of items per bucket.
- With small load factors (< 2/3, say), Hash Tables give average performance of $O(1)$ for insert, delete, and retrieve.
- Traversing can be a little slow for Hash Tables.
  - Typically $O(n + b)$ [$b$ is the number of buckets] for unsorted traverse.
  - Use an auxiliary Doubly Linked List for linear-time traverse.

Rehashing

- A Hash Table needs to be remade if the load factor gets too high.
- This can be expensive.

Thoughts

- Average-case performance of a Hash Table can be very good.
  - Significantly better than for a balanced search tree.
- Worst-case performance and rehashing overhead can be problems.
- Use Hash Tables 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.

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”. 😊
  - Say “TREE”. 😊 I’ve heard “TRY”. 😊
  - Ick.

Example

```
<table>
<thead>
<tr>
<th></th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>dig</td>
<td>0dig</td>
</tr>
<tr>
<td>dog</td>
<td>1og</td>
</tr>
<tr>
<td>dot</td>
<td>2t</td>
</tr>
<tr>
<td>dote</td>
<td>3e</td>
</tr>
<tr>
<td>doting</td>
<td>3ing</td>
</tr>
<tr>
<td>eggs</td>
<td>0eggs</td>
</tr>
</tbody>
</table>
```

Not a Prefix Tree!
A **Prefix Tree** (or Trie) is a tree that holds a list of sequences.

- It is space-efficient when many of those sequences share the same first few items.
- The quintessential example is **words**, as in the previous slide.

In a Prefix Tree for storing words, each node has 26 child pointers and a Boolean value.

- The child pointers correspond to the letters of the alphabet.
- The Boolean value indicates whether this node represents a valid word.

The Prefix Tree to the right holds 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 PrefixTreeNode {
    bool isWord;                  // true if a word ends here
    (PrefixTreeNode *) ptrs[26];  // a .. z ptrs; NULL if none
};
```

- Another possibility:

```c++
struct PrefixTreeNode {
    bool isWord;
    std::map<char, PrefixTreeNode *> ptrs;
};
```

*Class template std::map is an STL Table implementation.*

- Think “Red-Black Tree”.

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

Efficiency

- Retrieve, insert, and delete all take a number of steps proportional to the word length.
- This is logarithmic in the number of possible words.
- If word length is considered fixed, then it is constant time.

A Prefix Tree is a good basis for a Table implementation.

- The words in the list are the keys.
- The Prefix Trees we looked at are essentially a set of words.
- Add extra data to each node (or perhaps a pointer to extra data), and we could do key-based look-up.

A Prefix Tree works well with many kinds of lists of sequences:

- Short-ish sequences from a not-too-huge alphabet. Initial segments are shared.
- Examples: words in a dictionary, ZIP codes, etc.

The idea behind Prefix Trees has been used in other data structures.

- For example, you may want to look into Judy Arrays.
We now take a brief look at Table usage in various languages, beginning with C++.

- C++ STL
  - Sets: \texttt{std::set}
  - Maps: \texttt{std::map}
  - Other Tables
  - Algorithms for sets
- Other Languages
  - Python
  - Perl
  - Lisp
The C++ STL contains a “pair” template: `std::pair`, in `<utility>`. It acts as if it is declared roughly like this:

- Note the public data members.

```
template<typename T, typename U>
struct pair {
    T first;
    U second;
};
```

Other members, including `operator<` and `operator==`, exist.

- So you can put a `std::pair` into a sorted container, as long as types `T` and `U` have `operator<`.
- You can also do `std::find` (Sequential Search) to look for a `std::pair`, as long as types `T` and `U` have `operator==`. 
Tables in Practice
Aside: `std::pair` [2/2]

Example:

```cpp
std::pair<int, double> p;
p.first = 3;
p.second = 4.5;
// Or we can do the above using a ctor:
std::pair<int, double> p2(3, 4.5);
```

What do we use a `std::pair` for?

- To return more than one value from a function.
  - Remember `fibo3.cpp`?
- To return a range as two iterators.
  - As in `std::equal_range`, a variation of Binary Search.
- To store a key-data pair.
  - For use in key-based look-up.
The simplest STL Table implementation is `std::set`, in `<set>`.

- The key type and value type are the same.
  - That is, the key is the whole data item.
- Duplicate (equivalent) keys are not allowed.
  - Thus, all you can say about a value is whether or not it is in the structure.
  - In short, it is just what it says it is: a set.
- The specification was put together with a balanced search tree in mind. Most implementations will probably use a Red-Black Tree.

Declare a set as follows:

```cpp
std::set<valuetype> s;
```

An optional template parameter specifies the comparison used.

- This is done just as for sorting, Heap algorithms, etc.
- The default is to use `operator<`.

```cpp
std::set<valuetype, comparison> s;
```
A `std::set::iterator` is a bidirectional iterator.

- Items appear in sorted order.

A `std::set::iterator` is not a mutable iterator, that is, one cannot do "*iter = v;".

- Why not?
  - Because items are stored in sorted order. Changing an item might break this invariant.

Iterators and references are valid until the item is erased.

- How would this be implemented?
  - Clearly, references stay valid, because this is a node-based structure.
  - A Red-Black tree can be reorganized by an insertion or deletion. Thus, iterators must not store information about the structure of the tree (as they do in a `std::deque`).
  - So we must be able to find our way around the tree starting at a leaf. This means the tree must have parent pointers.
  - Conclusion: Give the tree parent pointers, and make the iterator a wrapper around a pointer.
Tables in Practice  
*std::set — Major Operations [1/2]*

Insert

- Given an item.
- Inserts given item into the set. Does nothing if an equivalent item (key) is already in the set.
- Returns a `std::pair<iterator, bool>`. The iterator points to the inserted item or the already present item. The bool is `true` if the insertion happened.

- Example:

```cpp
std::set<int> s;
s.insert(3);
if (!s.insert(4).second)
  cout << "4 was already present" << endl;
```
Erase
  • Given a key or an iterator.
  • Removes the proper item (if any) from the set.
  • Examples:

```cpp
s.erase(3);
s.erase(s.begin());
```

Find
  • Given a key.
  • Returns an iterator, which either points to the item or is end().
  • Example:

```cpp
If (s.find(3) != s.end())
  cout << "3 was found" << endl;
```

  • Why not just use std::find or std::binary_search?
    ▪ The former is always linear time. The latter is only efficient on random-access data.
There are many other members in `std::set`, including range insert & erase, etc.

One interesting member function is “insert with hint”.

- This works like regular insert, but it is given an iterator, too. It returns an iterator to the item.
- The second parameter (iterator) is a “hint” as to where the item should be inserted.
- The code *may* ignore the hint, but it probably uses it.
- How do you think this is typically implemented?
  - Probably the inorder traversal property of a Red-Black Tree is used to look for locations “near” the given one.
- What is a good hint to give?
  - If you are inserted items in sorted order, a good hint is the location of the last item inserted.
- What is the likely effect of giving a bad hint?
  - Slower behavior [but still $O(\log n)$, as the Standard requires].
The other main Table available in C++ is `std::map`, in `<map>`.

- The key and data types are specified separately.
- The value type is a pair: `std::pair<keytype, datatype>`.
- As with `std::set`:
  - Duplicate (equivalent) keys are not allowed.
  - The specification was put together with a balanced search tree in mind. The implementation is usually a Red-Black Tree.
  - An optional comparison can be specified. It defaults to using `operator<`.

Declaration:

```cpp
std::map<keytype, datatype> m;
```

or

```cpp
std::map<keytype, datatype, comparison> m;
```
Major operations in `std::map` are much the same as for `std::set`. A very convenient operation is: `data & operator[](key)`

- This allows a map to be used like an array. Examples:

```cpp
std::map<std::string, int> m;
m["abc"] = 7;
cout << m["abc"] << endl;
m["abc"] += 2;
```

- This is usually defined as follows ("k" is the given key):

```cpp
(*((m.insert(value_type(k, data_type()))).first)).second
```
More `operator[]` examples:

```c++
std::map<int, int> m2;
m[0] = 34;
m[123456789] = 28;  // Very little memory used!

std::map<std::string, std::string> id;
id["Hubert Gump"] = "abc";
cout << id["Fred Smurg"] << endl;
// The above line inserts
//     std::pair<std::string, std::string>
//                  ("Fred Smurg", std::string())
//     into the map. (Right?)
```
A map's `operator[]` is very convenient and useful. However ...

- This `operator[]` always inserts. Thus, it has no `const` version.

```cpp
void printEntry(const std::map<std::string, int> & m1) {
    cout << m1["abc"] << endl;  // DOES NOT COMPILE!
}
```

- Because of this insertion, `operator[]` is generally *not* a good way to check whether a given key is already in the map. Instead, use `map::find`.

```cpp
std::map<Foo, Bar> m2;
Foo theKey;
// I want to test whether theKey lies in m2
if (m2.find(theKey) != m2.end())  // GOOD way to test
if (m2[theKey] == ...)              // BAD way to test
```
Tables in Practice

std::map — Iterators

Iterators are much as for std::set.

- They are bidirectional iterators.
- Items appear in sorted order, by key.
- They are not mutable.
  - Thus, no "*iter = v;".

However, one can do "(*iter).second = d;".

- This makes sense, because the data part does not affect the organization of the structure.
- But how can we disallow the former, while allowing the latter?
- Answer: The value type is std::pair<const key_type, data_type>.

Note: This is okay, but we usually write iter->second = d;
The C++ STL also has `std::multiset` and `std::multimap`.
- These are much the same as `std::set` and `std::map`, except that duplicate (equivalent) keys are allowed.
- There is no `operator[]` in `std::multimap`.
- Retrieve operations typically involve either returning a range or else counting the number of matching items.
- See the doc’s.

- However, many STL implementations include `std::hash_set`, `std::hash_map`, `std::hash_multiset`, `std::hash_multimap`.
- These are nonstandard and (sadly) their interfaces vary a bit from one implementation to another.
- The upcoming revised C++ standard is expected to include Hash-Table-based classes.
The C++ STL contains special algorithms that deal with sorted sequences that may not be random-access: the set algorithms.

- These are `includes`, `set_union`, `set_intersection`, `set_difference`, and `set_symmetric_difference`.

Each of these takes two sorted sequences, each specified with two input iterators.

- Function `includes` returns a `bool`.
- For the others, the output is another sequence, written to a given output iterator (the 5th parameter).

How can we find the intersection of two sets and store the result as another set?

- It helps to use a special kind of iterator that does insertion.
The C++ STL offers several iterators that insert into containers.

- All are defined in the header `<iterator>`.

**A back insertion iterator** calls the `push_back` member function.

- Function `std::back_inserter` returns such an iterator. This takes one parameter: the container, which must have a `push_back` member.
- Say `v` is a `vector<int>`. Then `std::back_inserter(v)` returns an iterator. Doing “`*iter++ = item;`” with this iterator does “`v.push_back(item);`”

\[
\text{std::copy(iter1, iter2, std::back_inserter(v));}
\]

- The above inserts a copy of range `[iter1, iter2)` at the end of `v`.

There is also `std::front_inserter`, for doing `push_front`.

And there is `std::inserter`, which takes two parameters: a container `c` and an iterator `pos` into container `c`. It returns an iterator that does `pos = c.insert(item, pos);`

- Note: insert-with-hint for sets & maps looks like the above.
Again, how can we find the intersection of two sets, and store the result as another set?

- Use `std::set_intersection`. This takes two sorted sequences and writes the result, as a sorted sequence, to a given output iterator.
- For the inputs, use the begin & end member functions of the two given sets.
- For the output, use an inserter (special iterator) for a third set.

```cpp
#include <set>        // for std::set
#include <algorithm>  // for std::set_intersection
#include <iterator>   // for std::inserter

std::set<Foo> s1, s2, s3;
// Here, we initialize sets s1 and s2.
// Now, we want s3 to be the set intersection of s1, s2.
std::set_intersection(s1.begin(), s1.end(),
                      s2.begin(), s2.end(),
                      std::inserter(s3, s3.begin()));
```