More Data Structures

for PA#3, PA#4 and Project
Data Structures

• For sizeable programs, one problem that can quickly arise is that of data storage.
  – What is the most efficient or effective way to organize and utilize information within a program?
  – Quick answer – it depends on the task.
Data Structures

• For some tasks, it is helpful (at minimum) and possibly necessary to have sorted data.

• For other tasks, it is not necessary to note where any given piece of data is stored within a storage data structure.
Beyond Arrays

• In C++, there is a vector class as part of the std namespace.
  – Likewise, this class internally uses an array and resizes it when necessary as new items are added to the conceptual underlying list.
  – This resizing is also handled internally and automatically by the class.
Note that we now have two different ways of storing data, each of which has its own pros and cons.

- **Arrays**
  - Good for adding items to the end of lists and for random access to items within the list.
  - Bad for cases with many additions and removals at various places within the list.
Lists

• Note that we now have two different ways of storing data, each of which has its own pros and cons.
  – Linked Lists
    – Better for adding and removing items at random locations within the list.
    – Bad at randomly accessing items from the list.
      • Note that to use a random item within the list, we must travel the chain to find it.
Lists

• Note that both of these objects fulfill the same end goal – to represent a group of objects with some implied ordering upon them.

• While they meet this goal differently, their primary purpose is identical.
Beyond Lists

• We have this notion of a “list” structure, which maps its stored objects to indices.
  – What if we don’t actually need to have a lookup position for our stored objects?
  – But wait! How could we possibly iterate over the objects in a for loop?
The Iterator

• Many programming languages provide objects called *iterators* for enumerating objects contained within data structures.
  – C++ and Java are no exceptions.
  – C++’s versions are defined in the `<iterator>` header file.
The Iterator

• This iterator may be used to get each contained object in order, one at a time, in a controllable manner.
  – It’s especially designed to work well with for loops.
The Iterator

• Example code:

```cpp
vector<int> numbers;

// omitted code initializing numbers.

iterator<int> iter;
for(iter = numbers.begin(); iter != numbers.end(); iter++)
{
    cout << *iter << ' ';
}
```
The Iterator

• In C++, iterators are designed to look like and act something like pointers.
  – The * and -> operators are overloaded to give pointer-like semantics, allowing users of the iterator object to “dereference” the object currently “referenced” by the iterator.
In C++, iterators are designed to look like and act something like pointers.

- Furthermore, note the use of operator `++` to increment the iterator onto the next item.
- This is another way we can interact with pointers; it’s useful for iterating across an array while using pointer semantics.
The Iterator

vector<int> numbers;

// omitted code initializing numbers.

iterator<int> iter;
for(iter = numbers.begin(); iter != numbers.end(); iter++)
{
    cout << *iter << ' ';
}

The Iterator

- C++11 (the newest edition/standard) also provides an alternate version of the for-loop which is designed to work with iterable structures and iterators.

```c++
vector<Person> structure;
for(Person &p:structure)
{
    //Code.
}
```
The Iterator

- Both the `std::vector` and `std::list` classes of C++ implement iterators.
  - `begin()` returns an iterator to the list’s first element.
  - `end()` is a special iterator “just after” the final element of the list, useful for checking when we’re done with iteration.
Other Data Structures

• Let’s move on from this idea of a “list” structure.

• In particular, note how lists map their stored objects to indices.
  – What if we don’t actually need to have a lookup position for our stored objects?
  – In particular, does it really need to be an integer?
Other Data Structures

• There are many, many other techniques for storing data than the model of a list.
  – Such other data structures have different techniques for accessing and handling stored data.
  – These “different techniques” are often designed with a focus on different usage patterns.
Other Data Structures

• A first example: arrays index their contained objects by integers.
  – Should integers be the *only* thing by which we can index an item within a collection-oriented data structure?

apple  bear  A113  42  cake  blue  red  ...
Maps

• The interface built on this idea within Java is the `Map`.
  – `TreeMap` and `HashMap` are the two prominent implementations.
  – The `value` is the object being stored within the map.
  – The `key` for such a value is the data element used as an index into the map for that value.
Maps

• The classes built on this idea within C++ are `map` and `unordered_map`.
  – The `value` is the object being stored within the map.
  – The `key` for such a value is the data element used as an index into the map for that value.
Maps

• How would such a map work?
  – We could just use matching arrays for the keys and values.
  – However, this wouldn’t be the most efficient idea – better techniques are known.
Hash Maps

• *Hash* maps work by converting the key to a unique integer, where possible, through a *hashing function*.
  
  – C++: hash maps are represented by `unordered_map`.
  
  – The selection of such a function is not a simple operation.
  
  – As such, the constructor takes in a hashing function as an argument, mapping each key to a nearly-unique integer.
Hash Maps

• This “hash code” is then mapped into an array for storage.
  – Problem: the “hash code” can easily be larger than the storage array’s size.
  – Solution: modular arithmetic. Divide by the array’s size and use the remainder.
Hash Maps

New input: ("Harrison", "Harrison")

hash("Harrison")
-2070369658

-2070369658 mod 7
0

<table>
<thead>
<tr>
<th>i</th>
<th>Key</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>&quot;Harrison&quot;</td>
<td>&quot;Harrison&quot;</td>
</tr>
</tbody>
</table>
New input: ("Ford", "Ford")

\[
\text{hash("Ford")} \\
-2127646392 \\
\text{-2127646392 mod 7} \\
-4 \Rightarrow 3
\]
### Hash Maps

New input: ("Star", "Star")

hash("Star")

2068792

2068792 mod 7

5

<table>
<thead>
<tr>
<th>i</th>
<th>Key</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>“Harrison”</td>
<td>“Harrison”</td>
</tr>
<tr>
<td>1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>“Ford”</td>
<td>“Ford”</td>
</tr>
<tr>
<td>4</td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>“Star”</td>
<td>“Star”</td>
</tr>
<tr>
<td>6</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
New input: ("Wars", "Wars")

\[
\text{hash("Wars")} = -2026118662
\]

\[
-2026118662 \mod 7 = -1 \Rightarrow 6
\]
Hash Maps

• Pros:
  – direct, instant lookup of values, regardless of the key’s type.

• Cons:
  – does not support sorting
  – requires a specialized hashing function for keys that creates a unique int for each possible key.
Maps

• What if we want to have the entries sorted by their keys?
  – It is possible to build structures that efficiently keep their data *permanently* sorted by key!