Table ADT and Sorting

Algorithm topics continuing (or reviewing?) CS 24 curriculum
A table ADT (a.k.a. Dictionary, Map)

Table public interface:

// Put information in the table, and a unique key to identify it:
bool put(KeyType key, ItemType info);

// Get information from the table, according to the key value:
ItemType get(KeyType key);

// Update information that is already in the table:
bool update(KeyType key, ItemType newInfo);

// Remove information (and associated key) from the table:
bool remove(KeyType key);

// Above methods return false if unsuccessful (except get returns null)

// Print all information in table, in the order of the keys:
void printAll();
Table implementation options

- Many possibilities – depends on application
  - And how much trouble efficiency is worth
- Option 1: use a BST
  - To put: insertTree using key for ordering
  - To update: deleteTree, then insertTree
  - To printAll: use in-order traversal
- Option 2: use sorted array with binary searching
- Option 3: implement as a “hash table”
Hashing ideas and concepts

- **Idea:** *transform arbitrary key domain* (e.g., strings) into “dense integer range”
  - Then *use result as index* to table
  - `int index = hash(key);` // transform key to int

- **Collisions:** `hash(k1) == hash(k2), k1 != k2`
  - Usually impossible to avoid (“perfect hashing” rare)
  - Therefore, must have a way to handle collisions
    - e.g., if using “open addressing” techniques -
      ```c
      while (occupied(index)) index = probe(key);
      ```

- **Concept:** Insertion/searching is quick – but only until the table starts to get filled up
  - Then collisions start happening too often!
Implementing a hash table

- Constructor allocates memory for array of items, and initializes all items to “empty” key
  - size is size of array
  - n is the number of items in the table
  - Load factor is $n / \text{size}$

- put method uses $\text{hash(key)}$ (and $\text{probe(key)}$ if open address hashing) to find empty slot for new item
  - May be necessary to resize array
    - If so, also necessary to rehash existing items
    - If open address hashing, resize when load factor > 50%
Open address hashing

- get & update methods use hash(key) and probe(key) in exact same sequence as put
  - To find existing info where it was put
- remove is more complicated
  - Cannot just remove an item – future probes for get and update might terminate prematurely at empty slot
    - Common trick is to have “deleted” key
      - Problem with that is table can seem full prematurely
    - Inefficient alternative rehashes all items when any removed
- Note: to printAll in key order – must sort first
  - So $O(n \log n)$ at best!
Hash functions

● Goal: **uniform distribution** of keys
  – Means each index of table is equally likely
  – Important for reducing collisions

● Common approach is a *restricted transformation*
  – Step 1 – *transform* key to *large* integer
  – Step 2 – *restrict* integer to 0…size-1
    ● Usually done with modulus operator - %

● Lots of variations – partly depends on key type
  – General observation: hard to find a good hash function
  – Note: should be “cheap” to compute too – e.g., division is slower on most CPUs than addition
Resolving collisions

- Simplest open address approach is linear probing
  - If \(\text{index} = \text{hash(\text{key})}\) is not empty, try \(\text{index} + 1\), then \(\text{index} + 2\), …, until empty slot
    - Note: searching for first “open address”
  - Leads to “primary clusters” – collisions bunch up
- Quadratic probing – vary probe, like 1, 3, 6, …
  - Leads to “secondary clusters” but not as quickly
- Double hashing – \(\text{probe(\text{key})}\) varies by key
  - Best open addressing approach for avoiding clusters
- Or completely different approach – “chaining”
Chaining

- Constructor allocates memory for **array of Lists**, and creates an empty list for each element of the array
- **put** method uses **hash(key)** and appends to end of list at that index of array
  - Still should resize when load factor approaches 80%
    - Clustering is not a problem, but long lists slow performance
- **remove** method is easier now – just delete from list
- But lots more overhead than open addressing
  - Must store node links as well as key and info
  - Use list method calls instead of direct array access
Sorting

- Probably *the* most expensive common operation
- Problem: **arrange** \( a[0..n-1] \) by some ordering
  - e.g., in ascending order: \( a[i-1] \leq a[i], \ 0 < i < n \)
- Two general types of strategies
  - Comparison-based sorting – includes most strategies
    - Apply to any comparable data – (key, info) pairs
    - Lots of simple, inefficient algorithms
    - Some not-so-simple, but more efficient algorithms
  - Address calculation sorting – rarely used in practice
    - Must be tailored to fit the data – not all data are suitable
Selection sort

- Idea: build sorted sequence at end of array
- At each step:
  - Find largest value in not-yet-sorted portion
  - Exchange this value with the one at end of unsorted portion (now beginning of sorted portion)
- Complexity is $\mathcal{O}(n^2)$ – but simple to program
- Also best way to find kth largest, or top k values
Heap sort

- Another priority queue sorting algorithm
  - Note about selection sort: unsorted part of array is like a priority queue – remove greatest value at each step
  - Also recall that heaps make faster priority queues

- Idea: create heap out of unsorted portion, then remove one at a time and put in sorted portion

- Complexity is $O(n \log n)$
  - $O(n)$ to create heap + $O(n \log n)$ to remove/reheapify

- Note proof: $O(n \log n)$ is the fastest possible class of any comparison-based sorting algorithm
  - But constants do matter – so some are faster than others in practice
Insertion sort

- Generally “better” than other simple algorithms
- Inserts one element into sorted part of array
  - Must move other elements to make room for it

- Complexity is $O(n^2)$
  - But runs faster than selection sort and others in its class
  - Really quick on nearly sorted array

- Often used to supplement more sophisticated sorts
Divide & conquer strategies

- Idea: (1) divide array in two; (2) sort each part; (3) combine two parts to overall solution
- e.g., `mergeSort`
  ```java
  if (array is big enough to continue splitting) →
  divide array into left half and right half;
  mergeSort(left half);
  mergeSort(right half);
  merge(left half and right half together);
  else → sort small array in a simpler way
  ```
  - Cost each time to merge two halves is $O(n)$,
    and overall complexity is $O(n \log n)$
  - But notice it also uses $2n$ space
  - Commonly used to sort large files (i.e., when there are too many records to load all of them into memory at once)
Quick sort

- Invented in 1960 by C.A.R. Hoare
  - Studied extensively by many people since
  - Probably used more than any other sorting algorithm

- Basic (recursive) quicksort algorithm:
  
```java
if (there is something to sort)
{
    partition array;
    sort left part;
    sort right part;
}
```

- All the work is done by partition (a.k.a. split) function
- And there is no need to merge anything at the end
Partitioning (for quickSort)

- Arrange so elements in the two sub-arrays are on correct side of a pivot element
  - Also means pivot element ends up in its final position

\[
\begin{array}{c}
\text{pivot} \\
\text{all } \leq \text{ pivot} \quad \text{red} \quad \text{all } \geq \text{ pivot}
\end{array}
\]

- Done by performing two series of “scans”
  
  \[
  \begin{align*}
  \text{scan from (} i = \text{ left}) \text{ until } a[i] & \geq \text{ pivot;} \\
  \text{scan from (} j = \text{ right}) \text{ until } a[j] & \leq \text{ pivot;} \\
  \text{swap } a[i] \text{ and } a[j], & \text{ and continue both scans;} \\
  \text{stop scanning when } i & \geq j;
  \end{align*}
  \]
Quick sort (cont.)

- **Complexity is** $O(n \log n)$ **on average**
  - Fastest comparison-based sorting algorithm
  - But overkill, and not-so-fast with small arrays
    - Um … what about a small partition?!
  - One optimization applies insertion sort for partitions smaller than than 7 elements
- **And worst case is** $O(n^2)$ !
  - Depends on initial ordering and choice of pivot
- **Btw:** C library `qsort`, and C++ STL `sort`
## Compare 3 table implementations

<table>
<thead>
<tr>
<th>Table operation</th>
<th>Hash table</th>
<th>BST</th>
<th>Sorted array</th>
</tr>
</thead>
<tbody>
<tr>
<td>create (new table)</td>
<td>$O(n)$</td>
<td>$O(1)$</td>
<td>$O(n)$</td>
</tr>
<tr>
<td>get, update</td>
<td>$O(1)$</td>
<td>$O(\log n)$</td>
<td>$O(\log n)$</td>
</tr>
<tr>
<td>put</td>
<td>$O(1)$</td>
<td>$O(\log n)$</td>
<td>$O(n)$</td>
</tr>
<tr>
<td>remove</td>
<td>$O(1)$</td>
<td>$O(\log n)$</td>
<td>$O(n)$</td>
</tr>
<tr>
<td>printAll</td>
<td>$O(n \log n)$</td>
<td>$O(n)$</td>
<td>$O(n)$</td>
</tr>
</tbody>
</table>

- Conclusion: choice depends on table purpose and size of $n$
- Q. Ever want to use a sorted array?
  - A. It *depends*!