CS3230 Chapter 4 - Data Structures

Based on lectures by Chang Ee-Chien
Notes taken by Andrew Tan
AY18/19 Semester 1

These notes are not endorsed by the lecturers, and I have modified them (often significantly) after lectures. They are nowhere near accurate representations of what was actually lectured, and in particular, all errors are almost surely mine.

1 Data structures

A data structure is simply an organized method of storing and retrieving data.

Here we will look at 3 different data structures: priority queues, binary search trees, and hash tables.

1.1 Priority Queue - Heap

A **priority queue** is a data-structure that maintains a collection of elements and supports the following three operations:

- Insert: Insert an element
- ExtractMin: Remove the smallest element
- ReduceKey: Reduce the value of one element

A **heap** is a tree where any parent is smaller or equal to its children. The heap will allow us to implement a priority queue with its operations performing efficiently.

The heap itself is implemented as a binary tree, which is itself implemented with an array, where

- The parent of $A[i]$ is $A[\lfloor i/2 \rfloor]$
- The left child of $A[i]$ is $A[2i]$
- The right child of $A[i]$ is $A[2i+1]$

The element without a parent is the root. Thus, $A[1]$ is the root. Furthermore, an element without children is a leaf.

Thus, let A be an array of n elements representing a binary tree. A is a heap if

$$A[\lfloor i/2 \rfloor] \leq A[i] \text{ for all } 2 \leq i \leq n$$

The two basic operations on the heap are *SiftUp* and *SiftDown*. SiftUp and SiftDown allows the array to be modified if a value in the heap is changed to a smaller or larger value respectively, so that it is again a heap.

Iterative version of SiftUp, where A is the array, i is the value to SiftUp.

```
SiftUp(A, i):
   if i == 1 then return
   j ← ⌊i/2⌋; # where j is the parent
   if A[j] <= A[i] then return
   swap (A[i], A[j])
   SiftUp(A, j)
```

Iterative version of SiftDown, where A is the array, i is the value to SiftDown, n is the size of the heap.

```
SiftDown(A, i, n):
   j ← 2i
   while j <= n:
      if j < n and A[j + 1] < A[j]:
         j ← j + 1
      if A[i] <= A[j] then return;
      swap (A[i], A[j])
      i ← j
      j ← 2i
```

To implement a priority queue, we can use SiftUp and SiftDown:

```
insert(H, key):
   H[heap.size + 1] ← key
   SiftUp(H, heap.size + 1)
   heap.size++
```

```
extractMin(H):
   v ← H[1]
   H[1] ← H[heap.size]
   heap.size ← heap.size − 1
   SiftDown(H, heap.size, 1)
   return v
```

The complexity of SiftDown and SiftUp are in $O(\log(n))$, where n is the total number of elements in the array. Therefore, the complexity of insert and extractMin are in $O(\log(n))$

1.1.1 Heap sort

With the priority queue, we can now implement HeapSort:

```
HeapSort(A, n):
   Let H be the empty heap
   for i ← 1 to n:
      insert(H, A[i])
   for i ← 1 to n:
      A[i] ← extractMin(H)
```

The running time of HeapSort is:
$T(1) + T(2) + \cdots + T(n-1) + T(n)$
$+K(n) + K(n-1) + \cdots + K(2) + K(1)$
$< nT(n) + nK(n)$
where $T(n)$ and $K(n)$ are the running times of insert and extractMin respectively.

Thus, the running time of HeapSort is in $O(n \log(n))$

1.1.2 Make heap

We can sort n elements in decreasing order without creating a temporary array by making a heap from the input array and then swapping accordingly:

```
MakeHeapTopDown(A, n):
   for i ← 2 to n:
      SiftUp(A, i)
```

The idea is that we start off with a heap, and swap the top of the heap (i.e. the smallest element) with the last element of the array, and sift down accordingly. This splits the array into two sections: the heap and the sorted array.

```
HeapSortDecreasing(A, n):
   MakeHeap(A, n
   for i ← 1 to n:
      swap(A[1], A[n − 1 + i])
      SiftDown(A, n − i, i)
```

We can also MakeHeap recursively, by heapifying the left and right subtree recursively, and then perform a SiftDown.

Let $T(n)$ be the running time of MakeHeap, where n is the number of elements. For simplicity, assume $n = 2^k + 1$ for some k.

$$T(n) = 2T(\tfrac{n-1}{2}) + f(n) \text{ where } f(n) \in \Theta(\log(n))$$

Approximating the above by

$$T(n) = 2T(\tfrac{n}{2}) + f(n) \text{ where } f(n) \in \Theta(\log(n)),$$

we can use the Master Theorem to obtain $T(n) \in \Theta(n)$

MakeHeap also works from the bottom up, as such:

```
MakeHeapBottomUp(A, n):
   for i ← ⌊n/2⌋ to 1:
   SiftDown(A, i, n)
```

The number of comparisons here is less than

$$c \sum_{j=1}^{k} j 2^{k-j}$$

where c is some constant, $k = \log(n)$, and j represents the number of comparisons in a sub-tree at 'level j', and 2^{k-j} represents the number of sub-trees at that level.
Simplifying the above, we obtain a running time in $\Theta(n)$.

2 Binary search tree

A binary tree is a **binary search tree** if the values contained in every vertex is larger than or equal to the values contained in its left-subtree, and less than or equal to the values contained in its right subtree.

2.1 Tree search

Tree search is as simple as recursing into the appropriate subtree until we find the key.

2.2 Ordered traversal

There are multiple ways to systematically traverse a binary tree:

Given a binary tree root T, we recursively traverse the tree and print accordingly:

Pre-order:

- print current node
- traverse left subtree
- traverse right subtree

In-order:

- traverse left subtree
- print current node
- traverse right subtree

Post-order:

- traverse left subtree
- traverse right subtree
- print current node

2.3 Deletion in a BST

When deleting a node z from a BST T, the updated tree must still be a BST.
If z has no children, then the problem is trivial.
If z has one child, then we simply have the parent of z point to the child of z.
If z has two children, then we must first find the successor v of z, and replace z with v.

2.4 Balanced BST - red-black tree

A binary search tree supports deletion, insertion, and searching in $O(h)$ time, where h is the height of the tree.
The largest possible h is in $\Theta(n)$ where n is the total number of elements.
If we keep the binary tree balanced (also called a **red-black tree**), that is, for any node, the difference in height between the left and right subtree is at most one, then h is in $\Theta(\log(n))$.

3 Hash table

A hash table is a data structure that supports insertion, deletion, and searching, where, under reasonable assumptions, the expected time is in $\Theta(1)$.

Let U be the set of all possible keys, instead of creating an array of size $|U|$, we create a hash table H of a smaller size, say M.
A hash function h, is a function from U to $\{0, \dots, M-1\}$ (which we assume can be computed in $O(1)$, and a key k is stored in a slot $h(k)$.

3.1 Collision

If we wish to hash a key k in a slot $h(k)$ that is already occupied, we say that a **collision** occurred. **Collision resolutions** are methods to resolve such collisions.

3.1.1 Resolution 1 - Separate chaining

Instead of having each index of the hash table store a key, we have them store a linked list of keys, and append them accordingly.

Suppose a hash table H has M slots, and it stores n elements. We define the **load factor** for H as (n/M). We assume **simple uniform hashing**, that is, any given key is equally likely to hash into any of the M slots, independent of where any other element has hashed to.

Thus, the expected number of keys hashed to a particular slot is a, where a is the load factor. Therefore, the expected total time required to search for a key k is $O(1+a)$.

3.1.2 Resolution 2 - Open addressing

When collision occurs, we store the key in another available slot in the hash table. There are several methods to choose the available slot:

A **Linear Probing** When collision occurs, store the key in the next available slots greater than $h(k)$. That is, search through $h(k)+1, h(k)+2, \dots$.

B **Quadratic Probing** When collision occurs, store the key in the next available slot according to this sequence: $h(k)+1^2, h(k)+2^2, \dots$.

C **Double Hashing** Let $h(k,i)$ be a hash function with two variables. When collision occurs, store the key in the next available slot according to this sequence: $h(k,0), h(k,1), h(k,2), \dots$.

Given a key, the expected number rof probes required to know that key is not in the hash table is at most $1/(1-a) \approx 1 + a + a^2 + a^3 + \dots$
Likewise, given a key in the hash table, the expected number of probes required to find the location of the key is at most $(1/a)\ln(1-a)^{-1}$

A Graph and tree definitions

- A **directed graph** is a pair $G = \langle V, E \rangle$ where V is a set of **vertices** and $E \subseteq V \times V$ is a set of **edges**. If $(v, w) \in E$ then we say there is an edge from vertex v to vertex w.
- A **path** from v to u is a sequence of edges leaving from v to u.
- A **rooted tree** is a directed graph where there exists a vertex r such that every other vertex can be reached from r by a unique path. Hence r is denoted the **root**.
- In a tree, if (v, w) is an edge, we say that v is the **parent** of w, and w is the **child** of v.
- If there is a directed path from v to w, we say that w is the **descendant** of v.
- If a vertex has no children, it is a **leaf**. Otherwise, it is an **internal node**.
- If every vertex in the rooted tree has at most n children, then we say that this tree is a **n-ary tree**.
- The **height** of the tree is the number of edges along the longest path from the root to the leaves.
- A **binary tree** is a 2-ary tree. The two children of a binary tree are the **left-hand** child and **right-hand** child.
- The **predecessor** of a node p is the node occurring immediately before p in the in-order traversal of the tree.
- The **successor** of a node p is the node occuring immediately after p in the in-order traversal of the tree.