Data Structures and Algorithms Final

2017-04-25

- **Heap**: a *complete binary tree* each of whose paths follows *total order relation*.
    - Heapify: $O(n)$
    - Access extremum: $O(1)$
    - Remove extremum: $O(\log(n))$
    - Insert: $O(\log(n))$
- Heapify:

```
def Max_Heapify(A, i):
    l = left(i)
    r = right(i)
    largest = i
    if l <= A.size and A[l] > A[largest]:
        largest = l
    if r <= A.size and A[r] > A[largest]:
        largest = r
    if largest != i:
        swap(A[i], A[largest])
        Max_Heapify(A, largest)
```

- **Hash table**: given a hash table with m slots that stores n elements
    - **Key density**: n/T where T is the number of distinct possible keys.
    - **Load density (load factor)**: n/m.
    - **Hash function**: $h(k)$ is a function which maps a key to an indexing value.
    - If $h(k_1) = h(k_2)$, then k_1 and k_2 are said to be *synonyms* with respect to h.
- **Uniform hashing**:
    - Any given element is equally likely to hash into any of the m slots.
    - The probe sequence of each key is equally likely to be any of the $m!$ permutations.
- Some issues related to hashing are **collision** and **overflow** which can be made up for by **open addressing** and **chaining**.
- **Open addressing**:
    - All elements occupy the hash table itself.
    - **Linear probing**: $h(k,i) = (h'(k) + c_1 i)\%m$, for $i = 0, 1, \ldots, m-1$.
        - # probing sequence = $m \rightarrow$ **primary clustering**.
    - **Quadratic probing**: $h(k,i) = (h'(k) + c_1 i + c_2 i^2)\%m$, for $i = 0, 1, \ldots, m-1$.

        - # probing sequence = m → **secondary clustering**.
    - **Double hashing**: $h(k,i) = (h_1(k) + i \cdot h_2(k))\%m$.
        - # probing sequence = m^2 → the best design to approximate *uniform hashing*.
    - **Rehashing**: $h(k,i) = h_1 \circ h_2 \circ \ldots \circ h_{i-1} \circ h_i(k)$
    - $h(k_1,i) = h(k_2,i)$ iff $h(k_1,0) = h(k_2,0)$
    - The expected number of probes in an *unsuccessful* search is at most $\frac{1}{1-\alpha}$.
    - The expected number of probes in a *successful* search is at most $\frac{1}{\alpha}\ln(\frac{1}{1-\alpha})$.
    - The worst-case for an *unsuccessful* or a *successful* search is $O(n)$.
- **Chaining**:
    - All the elements that hash to the same slot are place into the same chain.
    - An *unsuccessful* search takes average-case time $\Theta(1+\alpha)$.
    - A *successful* search takes average-case time $\Theta(1+\alpha)$.
    - If m is at least proportional to n, i.e. $\alpha = O(1)$, searching takes *constant time* on average.
- Hash function designs:
    - **Division method**: $h(k) = k\%m$
    - **Multiplication method**: $h(k) = \lfloor m(kA\%1) \rfloor$, where $0 < A < 1$
    - **Universal hashing**
    - **Mid-square method**: $h(k)$ is the middle r bits of k^2
    - **Folding method**: **shift folding** and **boundary folding**.
    - **Digit analysis**: choose the best hash function set for a given set of known keys.
- **Dynamic hashing (extendible hashing)**:
    - *Directory* method:
        - **Directory**: an array of pointers to chains that store actual keys.
        - **Global depth**: the number of bits of $h(k)$ used to index the directory.
        - **Local depth**: the number of least significant bits shared by all entries in the same chain. Always ≤ *global* depth.
        - The directory doubles the size if one of the chains overflows.
        - A chain splits when it overflows and its *local* depth < *global* depth.
    - *Directoryless* method or **linear hashing**:
        - r: the number bits of $h(k)$ used to index into the hash table
        - q: the slot that will spiit next.
        - **Overflow slots**: indexed using $h(k, r+1)$. Range: $[0, q-1]$ and $[2^r, 2^r + q - 1]$
        - **Active slots**: indexed using $h(k, r)$. Range: $[q, 2^r - 1]$
        - Allows for the expansion of the hash table one slot at a time.

        - Search: If $h(k, r) < q$, then search the chain that begins at slot $h(k, r+1)$. Otherwise, search the chain that begins at slot $h(k, r)$

- [x] Homework: [0425_reading]

2017-05-02

- [x] Homework: [dsa_2017_hw3_3.pdf]
- Solutions: [b00401062_hw3.pdf], [manager.c], [router.c]

2017-05-09

- **Disjoint sets**:
    - **Set** is a group of elements without ordering.
    - **Equivalence relation**: reflexive, symmetric and transitive.
    - **Equivalence class**: every element is a set satisfies equivalence relation.
    - Any two equivalence classes are disjoint, i.e. disjoint sets.
    - Operations on disjoint sets: *make*, *union*, *find*.
- Algorithms for m union-find operations on a set of n objects:
    - Weighted: *by size* v.s. *by height*.
    - Path compression: *two-pass* v.s. *one-pass*.

Algorithms	Find	Union	Total
Quick-find (array or linked list)	$O(1)$	$O(n)$	$O(mn)$
Quick-union (tree)	$O(n)$	$O(1)$	$O(mn)$
Weighted quick-union	$O(\lg n)$	$O(1)$	$O(m \lg n)$
Weighted quick-union with path compression	$O(\lg^* n)$	$O(1)$	$O(m \lg^* n)$

- Sorting in linear time:
    - **Counting sort**
    - **Radix sort**: *most significant digit (MSD) first* v.s. *least significant digit (LSD) first*.
    - **Bucket sort**

2017-05-16

- A **red-black tree** is a binary tree that satisfies the following red-black properties:
    - Every node is either red or black.

    - The root is black.
    - Every leaf (NIL) is black.
    - If a node is red, then both its children are black.
    - For each node, all simple paths from the node to descendant leaves contain the same number of black nodes.
- Lemma: A red-black tree with n internal nodes has height $\Theta(\lg(n))$.
    - A red-black tree with n internal nodes has height at most $2\lg(n+1)$.
    - A red-black tree with n internal nodes has height at least $\lg(n+1)$.
- Structure of a node:
    - Attributes: `p`, `left`, `right`, `key`, `color`
    - `T.nil` serves as the a NIL node and the parent of the root node.
    - `T.root` specifies the root node.
- Rotation: an operation on the tree to restore the balance of the tree.

```
def Left_Rotate(T, x):
    y = x.right
    x.right = y.left
    if y.left != T.nil:
        y.left.p = x
    y.p = x.p
    if x.p == T.nil:
        T.root = y
    elif x == x.p.left:
        x.p.left = y
    else: # x == x.p.right
        x.p.right = y
    y.left = x
    x.p = y
```

- Insertion:

```
def RB_Insert_Fixup(T, z):
    while z.p.color == RED:
        if z.p == z.p.p.left:
            y = z.p.p.right
            if y.color == RED:
                z.p.color = BLACK
                y.color = BLACK
                z.p.p.color = RED
                z = z.p.p
            else: # y.color == BLACK
                if z == z.p.right:
                    z = z.p
                    Left_Rotate(T, z)
                z.p.color = BLACK
                z.p.p.color = RED
                Right_Rotate(T, z.p.p)
```

```
        else: # z.p == z.p.p.right
    T.root.color = BLACK
```

- Deletion:

```
def RB_Delete_Fixup(T, x):
    while x != T.root and x.color == BLACK
        if x == x.p.left:
            w = x.p.right
            if w.color == RED:
                w.color = BLACK
                x.p.color = RED
                Left_Rotate(T, x.p)
                w = x.p.right
            if w.left.color == BLACK and w.right.color == BLACK:
                w.color = RED
                x = x.p
            else: # w.left.color != BLACK or w.right.color != BLACK
                if w.right.color == BLACK:
                    w.left.color = BLACK
                    w.color = RED
                    Right_Rotate(T, w)
                    w = x.p.right
                w.color = x.p.color
                x.p.color = BLACK
                w.right.color = BLACK
                Left_Rotate(T, x.p)
                x = T.root
        else: # x == x.p.right
    x.color = BLACK
```

2017-05-23

- Weights of weighted edges can be stored in
  - The value of the adjacency matrix.
  - The list node of the adjacency list.
- *Adjacency matrix* v.s. *Adjacency list*:

Graph	Adjacency matrix	Adjacency list
Space	$O(\|V\|^2)$	$O(\|V\| + \|E\|)$
Edge search	$O(1)$	$O(\|E\|)$
Adjacent vertex search	$O(\|V\|^2)$	$O(\|E\|)$

- **Breadth-first search (BFS)**: $O(|V| + |E|)$
  - **Shortest-path distance**: $\delta(s, v)$ is the minimum number of edges in any path from vertex s to vertex v

- Fields: `color`, π (parent vertex), `d` (shortest-path distance)

```
def BFS(G, s):
    for v in G.V - {s}:
        v.color = WHITE
        v.d = ∞
        v.π = None
    s.color = GRAY
    s.d = 0
    s.π = None
    Q = Queue()
    Q.enqueue(s)
    while not Q.empty():
        u = Q.dequeue()
        for v in G.adj[u]:
            if v.color == WHITE:
                v.color = GRAY
                v.d = u.d + 1
                v.π = u
                Q.enqueue(v)
        u.color = BLACK
```

- Lemmas: Let $G = (V, E)$ be a directed or undirected graph, and $s \in V$ be an arbitrary source vertex.
    - For any edge $(u, v) \in E$, $\delta(s, v) \leq \delta(s, u) + 1$.
    - Suppose that BFS is run on G. Then upon termination, for each vertex $v \in V$, the value $v.d$ computed by BFS satisfies $v.d \geq \delta(s, v)$.
    - Suppose that during the execution of BFS on G, the queue Q contains the vertices $\langle v_1, \ldots v_r \rangle$, where v_1 is the head of Q and v_r is the tail. Then,$v_r.d \leq v_1.d + 1$ and $v_i.d \leq v_{i+1}.d$ for $i = 1, 2, \ldots, r - 1$.
    - Suppose that vertices v_i and v_j are enqueued during the execution of BFS, and that v_i is enqueued before v_j. Then $v_i.d \leq v_j.d$ at the time that v_j is enqueued.
    - Suppose that BFS is run on G. Then, during its execution, BFS discovers every vertex $v \in V$ that is reachable from the source s, and upon termination, $v.d = \delta(s, v)$ for all $v \in V$. Moreover, for any vertex $v \neq s$ that is reachable from s, one of the shortest paths from s to v is a shortest path from s to $v.\pi$ followed by the edge $(v.\pi, v)$.
- **Depth-first search (DFS)**: $O(|V| + |E|)$
    - Fields: `color`, π (parent vertex), `d` (time of visiting), `f` (time of finishing)

```
def DFS(G):
    for v in G.V:
        v.color = WHITE
        v.π = None
    time = 0
    for u in G.V:
```

```
        if u.color == WHITE:
            DFS_Visit(G, u)
```

```
def DFS_Visit(G, u):
    time = time + 1
    u.d = time
    u.color = GRAY
    for v in G.adj[u]:
        if v.color == WHITE:
            v.π = u
            DFS_Visit(G, v)
    u.color = BLACK
    time = time + 1
    u.f = time
```

2017-06-06

- B-tree of minimum degree $t \geq 2$ has at $t-1$ keys and at most $2t-1$ keys.
- If $n \geq 1$, then for any n-key B-tree of height h and minimum degree $t \geq 2$, $h \leq \log_t \frac{n+1}{2}$.
- Searching in B-tree:

```
def B_Tree_Search(x, k):
    i = 1
    while i <= x.n and k > x.key[i]:
        i = i + 1
    if i <= x.n and k == x.key[i]:
        return (x, i)
    elif x.leaf:
        return None
    else:
        Disk_Read(x.c[i])
        return B_Tree_Search(x.c[i], k)
```

- Create an empty B-tree:
    - Only the root is allowed to have fewer than the minimum number $t-1$ of keys.

```
def B_Tree_Create(T):
    x = Allocate_Node()
    x.leaf = True
    x.n = 0
    Disk_Write(x)
    T.root = x
```

- Inserting a key into a B-tree:
    - Never step into a full node.
    - On inserting a key into a full node, the full node is splitted around its median key.
    - Splitting the root is the only way to increase the height of a B-tree.

```
def B_Tree_Split_Child(x, i): # split the i-th child of x
    z = Allocate_Node() # z will be the right sibling of y
```

```
    y = x.c[i] # y is the i-th child of x
    z.leaf = y.leaf
    z.n = t - 1
    for j = 1 ... t - 1:
        z.key[j] = y.key[j + t]
    if not y.leaf:
        for j = 1 ... t:
            z.c[j] = y.c[j + t]
    y.n = t - 1
    for j = x.n + 1 ... i + 1: # right shift x.c
        x.c[j + i] = x.c[j]
    x.c[i + 1] = z
    for j = x.n ... i: # right shift x.key
        x.key[j + 1] = x.key[j]
    x.key[i] = y.key[t]
    x.n = x.n + 1
    Disk_Write(y)
    Disk_Write(z)
    Disk_Write(x)
```

```
def B_Tree_Insert(T, k):
    r = T.root
    if r.n == 2t - 1:
        s = Allocate_Node()
        T.root = s
        s.leaf = False
        s.n = 0
        s.c[1] = r
        B_Tree_Split_Child(s, 1)
        B_Tree_Insert_NonFull(s, k)
    else: # r.n < 2t - 1
        B_Tree_Insert_NonFull(r, k)
```

```
def B_Tree_Insert_NonFull(x, k):
    i = x.n
    if x.leaf:
        while i >= 1 and k < x.key[i]:
            x.key[i + 1] = x.key[i]
            i = i - 1
        x.key[i + 1] = k
        x.n = x.n + 1
        Disk_Write(x)
    else: # not x.leaf
        while i >= 1 and k < x.key[i]:
            i = i - 1
        i = i + 1
        Disk_Read(x.c[i])
        if x.c[i].n == 2t - 1:
            B_Tree_Splid_Child(x, i)
            if k > x.key[i]:
                i = i + 1
        B_Tree_Insert_NonFull(x.c[i], k)
```

- Deleting a key from a B-tree:

- Never step into a minimal node.
- If the root node x ever becomes an internal node having no keys, then we delete x, and x's only child becomes the new root of the tree, decreasing the height of the tree by one.
- If the key k is in node x and x is a leaf, delete the key k from x.
- If the key k is in node x and x is an internal node, do the following:
  - If the child y that precedes k in node x has at least t keys, then find the predecessor k' of k in the subtree rooted at y. Recursively delete k', and replace k by k' in x.
  - If y has fewer than t keys, then, symmetrically, examine the child z that follows k in node x. If z has at least t keys, then find the successor k' of k in the subtree rooted at z. Recursively delete k', and replace k by k' in x.
  - Otherwise, if both $y$and z have only $t - 1$ keys, merge $k$and all of$z$ into y, so that x loses both k and the pointer to z, and y now contains $2t - 1$ keys. Then free z and recursively delete k from y.
- If the key k is not present in internal node x, determine the root $x.c_i$ of the appropriate subtree that must contain k, if k is in the tree at all. If $x.c_i$ has only $t - 1$ keys, execute step 3a or 3b as necessary to guarantee that we descend to a node containing at least t keys. Then finish by recursing on the appropriate child of x.
  - If $x.c_i$ has only $t - 1$ keys but has an immediate sibling with at least t keys, give $x.c_i$ an extra key by moving a key from x down into $x.c_i$, moving a key from $x.c_i$'s immediate left or right sibling up into x, and moving the appropriate child pointer from the sibling into $x.c_i$.
  - If $x.c_i$ and both of $x.c_i$'s immediate siblings have $t - 1$ keys, merge $x.c_i$ with one sibling, which involves moving a key from x down into the new merged node to become the median key for that node.

2017-06-13

- Polynomials:
  - **Degree**: highest order term with nonzero coefficient.
  - **Degree-bound**: any integer strictly greater than the degree.
  - Representation: coefficient v.s. point-value representation.
- Polynomial multiplication: Given two polynomials A and B with degree-bound n represented as coefficient vectors, and $C = A \times B$.
  - Extend the coefficient vectors of A and B size $2n$ by adding n zero coefficient to high-order terms.

- Evaluate the *discrete Fourier transform (DFT)* of the two coefficient vectors. $\rightarrow O(n \lg n)$
- Pointwise multiplication resulting in the *discrete Fourier transform (DFT)* of the coefficient vector of C. $\rightarrow O(n)$
- Interpolate back to the coefficient vector of C. $\rightarrow O(n \lg n)$

- **Discrete Fourier transform (DFT)**: $y_k = A(\omega_n^k) = \sum_{j=0}^{n-1} a_j \omega_n^{kj}$
  - $\omega_n = \exp(2\pi i/n)$
  - a is in frequency domain and y is in time domain.
- **Fast Fourier transform (FFT)**:

```
def FFT(a):
    n = a.length
    if n == 1:
        return a
    w = exp(2πi/n)
    a0 = [a[0], a[2], ..., a[n-2]]
    a1 = [a[1], a[3], ..., a[n-1]]
    y0 = FFT(a0)
    y1 = FFT(a1)
    for k in range(n/2):
        y[k] = y0[k] + w^k * y1[k]
        y[k + n/2] = y0[k] - w^k * y1[k]
    return y
```