Algorithms - 2 Data Structures

Data Structure

DS is a way to store and organize data in order to facilitate access and modifications

Basic Abstract Data Types (ADTs)

A mathematical model of the data methods used to modify and access the data, we don't care the implementation

Including: list, stack, queue, set,dictionary

stack

> PUSH(S,x), POP(S)

queue

> ENQUEUE(Q,x), DEQUEUE(Q)
> Q.head, Q.tail, Q.length
>
> **Priority queue**
>
> > data with a key
> > INSERT(P,x), EXTRACT-MAX(P), MAX(P), INCREASE-KEY(P,x,k)
> >
> > Applications: OS(manage jobs), Graph Algorithms(Dijkstra), arrange events, compression(Huffman encoding), heapsort...

> > How should we implement the priority queue:
> > binary heap,..

Common data structures

---

1. Heap

> **Binary Heap**
>
> > **structure**
> >
> > > array structure, visualized as nearly complete tree(only the left of last layer may be not full)
> > >
> > > root = A[1], PARENT(i) = $\frac{i}{2}$, LEFT(i)= $2i$, RIGHT(i)=$2i$+1
> > > attributes: A.length
> > >
> > > **Theorem: A nearly complete binary tree of n nodes has height** $\theta(logn)$
> > > prove:
> > >
> > > > a complete binary tree with height h has $\sum_{i=0}^{h} 2^i = \frac{1-2^{h+1}}{1-2} = 2^{h+1} - 1$ nodes
> > > > so for a nearly complete tree $1 + \sum_{i=0}^{h-1} 2^i = 2^h \leq n \leq 2^{h+1} - 1$
> > >
> > > Some other properties about the tree:
> > > https://cs.stackexchange.com/questions/841/proving-a-binary-heap-has-lceil-n-2-rceil-leaves
> >
> > **Attribute**
> >
> > > min binary heap PARENT(A[i]) $\leq$ A[i]
> >
> > **Operations** - e.g. Min-heap
> >
> > ```
> > HEAP-MINIMUM(A)
> >     return A[1]
> >
> > HEAP-EXTRACT-MIN(A)
> >   if A.heap-size < 1
> >   error 'heap underflow'
> >   min = A[1]
> >   A[1] = A[A.heap-size]
> >   A.heap-size = A.heap-size - 1
> >   MIN-HEAPIFY(A, 1)
> >   return min
> >
> > MIN-HEAP-INSERT(A, key)
> >   A.heap-size = A.heap-size + 1
> >   A[A.heap-size] = inf
> >   HEAP-DECREASER-KEY(A, A.heap-size, key)
> >
> > HEAP-DECREASE-KEY(A,i,key)
> >   if key > A[i]
> >   error 'new key is larger'
> >   A[i] = key
> >   while i > 1 and A[PARENT(i)] > A[i]
> >   exchange A[i] with A[PARENT(i)]
> >   i = PARENT(i)
> > ```

```
BUID-MIN-HEAP(A)
    A.heap-size = A.length
    for i = floor(A.length/2) downto 1
        MAX-HEAPIFY(A,i)

MIN_HEAPIFY(A,i)
    l = LEFT(i)
    r = RIGHT(i)
    if l <= A.heap-size and A[l] < A[i]
        smallest = l
    else smallest = i
    if r <= A.heap-size and A[r] < A[smallest]
        smallest = r
    if smallest != i:
        exchange A[i] and A[smallest]
        MIN-HEAPIFY(A, smallest)

HEAPSORT(A) #O(nlogn)
    BUILD-MAX-HEAP(A)
    for i = A.length downto 2
        exchange A[l] with A[1]
        A.heap-size == A.heap-size - 1
        MAX-HEAPIFY(A, 1)
```

Time complexity of build a heap: O(n)

> Amortized analysis: determine worst-case of a sequence of data structure operations
>
> | layer | height | depth | nodes | time = relate node*c |
> |---|---|---|---|---|
> | 1 | 3 | 0 | $1 = 2^0$ | $c \cdot 3 \cdot 2^0$ |
> | 2 | 2 | 1 | $2 = 2^1$ | $c \cdot 2 \cdot 2^1$ |
> | 3 | 1 | 2 | $4 = 2^2$ | $c \cdot 1 \cdot 2^0$ |
> | 4 | 0 | 3 | <=8 = 2^3 | |
>
> h is the height of the tree
> $T(n) \leq c\sum_{i=1}^{h} 2^{h-i} \cdot i \leq c \cdot s^h \sum_{i=1}^{h} s^{h-i} \cdot i \leq c \cdot 2^{logn} \sum_{i=1}^{h} 2^{-i} \cdot i \cdot \leq cn \sum_{i=1}^{\text{inf}} 2^{-i} \cdot i$
> $= cn \sum_{i=1}^{inf} (\frac{1}{2})^i \cdot i = \frac{cn}{2} \sum_{i=1}^{inf} i \cdot (\frac{1}{2})^{i-1} = \frac{cn}{2} \frac{1}{(1-1/2)^2} = O(n)$
>
> notice that:
> $\frac{\partial \sum_{i=0}^{inf} x^i}{\partial x} = \sum_{i=0}^{inf} i \cdot x^{i-1}$
> $\frac{\partial \frac{1}{1-x}}{\partial x} = \frac{1}{(1-x)^2}$
> $\sum_{i=0}^{inf} x^i = \frac{1}{1-x}$

2. Hash

dynamic Set operations:

> > key(x) = k(satellite data)
> > SEARCH(S,k)
> > MINIMUN(S)/MAXIMUM(S)
> > SUCCESSOR(S,x)
> > PREDECESSOR(S,x)
>
> **dictionary operation**
>
> > SERACH(S,k) - returns a pointer x, where x.key = k or nil if not found
> > INSERT(S,x)
> > DELETE(S,x)
>
> **dictionary data structure**
>
> > universe of keys U
> > a data structure that stores a subset $S \subseteq U$
> > operations:
> > SEARCH(S,k): given $k \in u$
> > INSERT(S,x): given x, a pointer to $k \notin S$
> > DELETE(S,x): given x, a pointer to $k \in S$

Hashing and Hash Tables

> > a hash table is an array T of size m
> > a hash function creates and index in the array from an $k \in U, h: U \rightarrow \{0, \ldots, m-1\}$ (hashes to slot h(k) in T)
> >
> > **Collision solution - chaining:**
> >
> > > Operations:
> > >
> > > > CHAINED-HASH-INSERT(S,x): insert x at the head of lst T[x.key]
> > > > CHAINED-HASH-DELETE(S,x): delete x from the list T[x.key]
> > >
> > > Assumption:
> > >
> > > > Simple uniform hashing: any key is equally likely to hash to any of the m slots
> > > > for all $i \neq j, pr[h(k_i) = h(k_j)] = \frac{1}{m}$
> > >
> > > Analysis for unsuccessful search
> > >
> > > > Let S be the items already in the table where $x \notin S, |S| = n$
> > > > $C_{x,y} = \begin{cases} 1 & h(x) = h(y) \\ o & otherwise \end{cases}$
> > > > $E[C_{x,y}] = 1 \cdot P[h(x) = h(y)] + 0 \cdot P[h(x) \neq h*(y)] = \frac{1}{m}$
> > > > $C_x = \sum_{y \in S. y \neq x} C_{x,y}$ = # of items that x has a collision with
> > > > $E[C_x] = \sum_{y \neq x} E[C_{x,y}] = \frac{n}{m}$ = # items in the table divided by the size of the table
> > > > So, if $n = O(m)$ then $\alpha = n/m = O(m)/m = O(1)$, for insertion, deletion, find

Universal Hashing:

> > **choosing a hash function**
> >
> > > using universal Hashing - randomness
> > > Solve:
> > > 1.Unifoma random
> > > 2.fundamental weakness

> > > for any hash function, we can find a set of keys that are hashed to the same spot
>
> > **Universal Class of Hash Functions:**
> >
> > > $H = h_1, h_2, \ldots, h_w$ is a universal hash function family if for all $k, k' \in U, Pr_{h \in H}[h(k) = h(k') \leq \frac{1}{m}]$, m is the table size
> >
> > **How to construct a universal Hash functions family:**
> >
> > > $H_{pm} = \{h_{a,b}(x) = ((ax+b)\ mod\ p)\ \}\ 1 \leq a \leq p-1, 0 \leq b \leq p-1,\ and\ p \geq all\ keys$
> > >
> > > Proof:
> > > Let $k, l \in U$ are two keys, WLOG $k > l, k \neq l$.
> > >
> > > 1. **If $h_{ab}(k) = h_{ab}(l)$ it is because they collided after mod p or mod m   - Property of prime numbers**
> > >   p is a prime larger than any key, and p > m
> > > 2. **If $h_{ab}(k) = h_{ab}(l)$ it is because they collided after mod m - Modulo arithmetic**
> > >   let r = (ak+b) mod p, s = (sl + b) mod p
> > >   since a < p,   k,l <p => a doesn't divide p, (k-l) doesn't divide p
> > >   the collision cannot be mod p since o/w $0 \neq s - r \equiv a(k-l)\ mod\ p$
> > > 3. **There is a 1-1 correspondence between the (r,s) pairs and the (a,b) pairs**
> > >
> > >   a = ((r-s)((k-l)^-1 mod p)) mod p
> > >   b = (r-ak) mod p
> > >   There are p(p-1) possible pairs (r,s) that $r \neq s$, thus there is a 1-1 correspondence between pairs (a,b) and pairs (r,s). Thus if a, b is chosen randomly the pair (r,s) is equally likely to be any pair of distinct values modulo p
> > > 4. **If we randomly choose an (r,s) pair, the probability that they collided mode m is <= 1/m.** Thus if I randomly choose an (a. b) pair the probability that $h_{a,b}(k) = h_{a,b}(l)$ is at most $\frac{1}{m}$.
> > >   # of choice for s such that $r \neq r$ and $s \equiv r$ is at most $\lceil \frac{p}{m} \rceil - 1$
> > >   (just list all the possible s that collide with r: r'= r mod m
> > >   r', r'+m, r'+2m, r'+3m, r'+4m, ..., r' + $(\lfloor \frac{p}{m} \rfloor - 1)m)$ are all less than p, r' + $(\lceil \frac{p}{m} \rceil - 1)m)$ may be less than p
> >
> > **Theorem**
> >
> > > If for all $k, k' \in U, Pr_{h \in H}[h(k) = h(k')] \leq \frac{1}{m}$, then for any $S \subseteq U$, for any $x \in U - S$ the expected number of collisions between x and the other elements in S is at most n/m, when |S|=n
> > >
> > > Proof:
> > >
> > > Corollary 1: If   for all $k, k' \in U, Pr_{h \in H}[h(k) = h(k')] \leq \frac{1}{m}$, then for any $S \subseteq U$, for any $x \in U - S$ thee expected number of comparisons for a successful search is O(1+n/m) and for an unsuccessful search is O(n/m) where |S| = n.
> > >
> > > Corollary 2: Using universal hashing and collision resolution by chaining in an initially empty table with m slots, it takes expected time $\theta(n)$ to handle any sequence of n INSERT, SEARCH, and DELETE operations containing O(m) INSERT operations where m is the table size

Perfect Hashing

> **Static**
>
> > We can do better when keys are static:
> > Given a fixed set of n keys we can construct a static hash table of size $m = \theta(n)$ such that search takes $\theta(1)$ time in the worst case
> >
> > > **Theorem:**
> > > Hashing n keys into $m = n^2$ slots using $h \in_R H$ then E(# collisions) < 1/2

Proof:

probability that two keys collide is $1/m = 1/n^2$, and # pairs of key = $\binom{n}{2}$

E(# collisions) $\leq \binom{n}{2} \cdot \frac{1}{n^2} = \frac{n(n-1)}{2} \cdot \frac{1}{n^2} < \frac{1}{2}$

Corollary:

Probability of no collisions > 1/2

Proof: X be a R.V. holding the number of collisions, according to Markov inequality:

$Pr[X \geq 1] \leq \frac{E[X]}{1} < \frac{1}{2}$

(FYI: Markov's inequality can refer to Math.md)

To reduce the space: two levels

Theorem:

Let m=n be the size of the hash table at level 1

Let n_j^2 be the size of the hash table at index j

Let h be randomly chosen from a universal set H

Then $Pr[\sum_{j=0}^{m-1}(n_j)^2 \geq 4n] < 1/2$

Proof:

the number of pairs that collide at the first layer:

$\sum_{x\in S}\sum_{y\in S} C_{x,y} = \sum_{j=0}^{m-1}\sum_{x\in S_j}\sum_{y\in S_j} C_{x,y} = \sum_{j=0}^{m-1}(n_j)^2$

$E[\sum_{j=0}^{m-1} n_i^2] = E[\sum_{x\in S}\sum_{y\in S} C_{xy}] = n + \sum_{x\in S}\sum_{y\in S-x} E[C_{xy}] \leq n + n(n-1)/m < 2n$

Markov's inequality: $Pr[\sum_{i=1}^{m-1}(n_i)^2 \geq 4n] < 2n/4n = 1/2$

dynamic

Consistent Hashing

Bloom-filer

is a space-efficient probabilistic data structure for test membership

property:

False positive - increase with n

never generate false negative result

can't delete element

can always adding element with the prize that FP increaser

Operations:

insert(x), lookup(x)

Probability of FP:

m: size of bit array,

k: number of hash functions

n: number of expected elements to be inserted in the filter

$P = (1-(1-\frac{1}{m})^{kn})^k$

3. Tree

> we may need more operations than dictionary
> each node: .key, .satellite_data, .p, .left, .right
>
> ### Self Balancing Tree
>
> > Balanced: height is O(logn)
> > 2-3, 2-3-4, Red-Black Trees, Augmenting Data
> > B-trees
> > 2-3 Tree
> >
> > > at most 3 children, all external nodes have same depth
> > > 2-node: 1 key 2 links
> > > 3-node: 2 keys 3 links
> > > $\lfloor log_3 n \rfloor \le h \le \lfloor log_2 n \rfloor$
> > >
> > > How to insert a key: local transformation preserve global property: 1. tree is ordered, 2. perfectly balanced
> > >
> > > > always insert into an existing node
> > > > always maintain depth condition
> > > > split node if a 4-node:
> > > >
> > > > > v, is created
> > > > > split into two
>
> ### Binary Search Tree
>
> > the left tree are all smaller, and the right trees are all greater

Red-Black Tree

> is a binary search tree that obeys **5 properties**:
>
> > 1.Every node is colored red or black
> > 2.The root is black
> > 3.Every leaf(T.nil) is black and doesn't contain any data
> > 4.Both Children of a red node are black
> > 5.For every node in the tree, all paths from that node down to a leaf(nil) have the same number of black nodes along the path
>
> Creating a Red-Black BST from a 2-3-4 tree Cont.
>
> Creating a 2-3-4 tree from Red-Black BST
>
> **Claim: The subtree rooted at a node, x, contains at least $2^{bh(x)}-1$ internal nodes (non nil)**
>
> > Basis: if x has height 0, it is a lef(nil node) and bh(x) = 0, the subtree rooted at x has 0 internal nodes $2^0 - 1 = 0$
> >
> > Inductive hypothesis: A node y of height at most h-1 contains at least $2^{bh(y)} - 1$.
>
> **Lemma: A red-black tree with n internal nodes has height at most 2lg(n+1)**

> Proof: Let h be the height, and b= bh(x) be the black height of the root node = x. By the previous two claims $n \geq 2^b - 1 \geq 2^{h/2} - 1$. Thus $n+1 \geq 2^b \geq 2^{h/2} \implies lg(n+1) \geq h/2 \implies h \leq 2lg(n+1)$

Corollary: On a red-black tree with n nodes, we can implement dynamic-set operations SEARCH, MINIMUM, MAXIMUM, PREDECCESOR.... In (logn)

Modify the Tree:

> Insert it as a red node, and then handle violation
> There are three possible violations:
> Case1: sibling of parent is red
> Case2:
> Case3:

```
LEFT-ROTATE(T,x)
    y = x.right
    x.righ = y.left
    if y.left != T.nil:
    y.left.p = x

RB-INSERT(T,z)
    // find where to insert
    y = T.nil
    x = T.root
    while x!= T.nil
        y = x
        if z.key < x.key
            x = x.left
        else: x = x.right
    // create node and attach it
    z.p = y
    if y == T.nil: // tree T was empty
        T.root = z
    else if z.key < y.key
        y.left = z
    else y.right = z
    // solve color violation
    z.left = T.nil
    z.right = T.nil
    z.color = RED
    RB-INSERT-FIXUP(T,z)

RB-INSERT-FIXUP(T,z)
    while z.p.color == RED
        if z.p == z.p.p.right: // parent is a left child
            y = z.p.p.left // y is uncle
            if y.color = RED // case 1: just change the color
                z.p.color = BLACK
                y.color = BLACK
                z.p.p.color = RED
                z = z.p.p
            else // case 2 or case 3
                if z == z.p.left // z is a left child case 2
                    z = z.p
                    RIGHT-ROTATE(T,z)
                // handle case 3
                z.p.color = BLACK
                z.p.p.color = RED
                LEFT-ROTATE(T,z.p.p)
        else (same as then clause with right and left exchanged)
```

```
T.root = Black
```

Proof we maintain the Red-Black tree properties:

> **Loop invariant:**
> At the start of each iteration of the while loop, node z is red.
> There is at most one red-black tree violation either:
> a. Z is a red root
> b. z and z.p are bother red
>
> **Initialization:**
> before we add the new red node z,
>
> Termination: loop terminates when z.p is black. Thus, there is no red-red violation.
>
> **Maintenance:**
> WLOG we only consider the 3 cases, where z.p is the right child of z.p.p let y = z.p.p.left // y is z's uncle. z.p.p is black, since z and z.p are red and there is only one red-red violation
>
> **Case1:** y is red and z.p.p is black, by making y and z.p black and z.p.p red. Thus black height property is maintained, but we might have created a red-red violation between z.p.p and its parent. Assign z = z.p.p
>
> **Case2:** y is black and z is a left child. Set z = z.p right rotate around z. Now z is now a right child.Both z and z.p are red. Only one case 3 red-red violation.
>
> **Case3:** y is black z is a right child, make z.p black and z.p.p red left rotate on z.p.p