- Binary Search Tree
- Hash Table

	Hashing	BST
- insert (k,v)	O(1)*	O(h)
- lookup (key)	O(1)*	O(h)
- delete (key)	O(1)*	O(h)

O(n)* worst case when resize is needed

h = log n

amortized running time

- BST

```
class Node {
    int key;
    Node left;
    Node right;
}
```

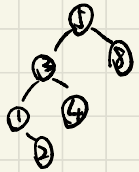
Definition of BST
1. left subtree is BST
2. right subtree is BST
3. root.key > all keys in the left subtree
   root.key < all keys in the right subtree

Tree Traversals
- level order
- pre order
- in order
- post order

```
levelOrder(Node root){
    Queue q;
    q.enqueue(root);        q.enqueue(root, 1);  wrap level with root
    while(!q.isEmpty()){
        Node cur = q.dequeue();          cur.level
        // processing on cur : print cur.key
        if(cur.left != null)
            q.enqueue(cur.left);         q.enqueue(cur.left, cur.level+1)
        if(cur.right != null)
            q.enqueue(cur.right);        q.enqueue(cur.right, cur.level+1)
    }
```

BST Right Side View

```
preorder (Node root){
  if(root == null) return;
  process(root);
  preorder(root.left);
  preorder(root.right);
}
```



preorder: 5, 3, 1, 2, 4, 8.
inorder: 1, 2, 3, 4, 5, 8
postorder: 2, 1, 4, 3, 8, 5

```
boolean lookup(Node cur, int key)                    O(h)
    if(cur == null) return false;
    if(cur.key == key) return true;
    if(cur.key < key) return lookup(cur.right, key);
    else return lookup(cur.left, key);
```

```
Node insert(Node cur, int key){                      O(h)
    if(cur == null) return new Node(key);
    if(cur.key == key) return cur; // already exist
    if(cur.key > key)   cur.left = insert(cur.left, key);
    else   cur.right = insert(cur.right, key);
    return cur;
}
```

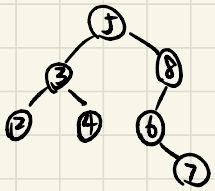
```
Node delete(Node cur, int key){
```

case 1 : No children : update parent's ref

case 2 : Has one child : parent pointing to single child

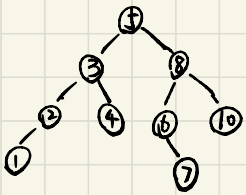
case 3 : Has two children :

- Find the predecessor / successor
- Swap
- delete recursively



1. delete 7
2. delete 8
3. delete 5

Find Predecessor



Case1. Rightmost node in left subtree O(h)
pred(8) : 7

Case2: First smaller on parent path
pred(6) : 5

```
class Node {
    int key;
    Node left
    Node right
    Node parent;
}
```

Case1.

```
Node predecessor(node){
    while(node.right != null)
        node = node.right
    return node
}
```

Case2

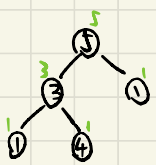
```
Node predecessor(node){
    cur = node;
    while( cur.parent != null )
        cur = cur.parent;
        if( cur.val < node.val )
            return cur;
    return null;
}
```

- Find kth smallest key in BST

#1 inorder traversal, stop at kth step O(n)* *worse case

#2 If we know the size of subtree, we can ignore half of the tree O(h)

```
Node {
    ...
    ...
    int size
}
```



Find 3th smallest
kth(5, 3).
kth(3, 3)
kth(4, 1) =>return.

```
kth(Node root, int k){
    rank = root.left.size + 1;
    if(rank == k) return root;
    else if(rank < k)
        return kth(root.right, k-rank);
    else
        return kth(root.left, k);
}
```

Hashing / HashTable ⇒ Generalization of arrays without indexing by integer.

- lookup
- insert
- delete

} O(1)*

1. prehash : convert any data type <string, float, int ...> into integer
2. hash : The integer we get on step 1 could be very large, we need to translate them into resonable range.

1.

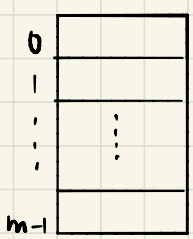
* If we have user defined data type . pair .
  We need to design prehash function ourself
  $\langle x, y \rangle = x * 41 + y$ // where 41 is a prime number

```
pair {
  int x
  int y
}
```

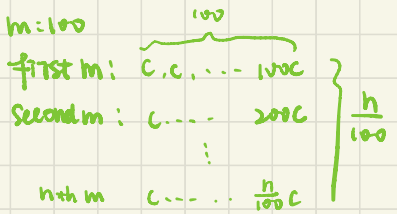
* String : using ASCII since each char is a 8-bit number in ASCII
  St = "hello" =
  St = "hell" = $'h' \times 41^3 + 'e' \times 41^2 + 'l' \times 41^1 + 'l' \times 41^0$

  when encounter a very long string
  Optimization :
  - Subset
  - cache

HashTable
key : Integer [in large space]
insert(key)
hash(key) ⇒ int ∈ [0, m)



m = 100



1. How to choose m ?
2. Hash function
3. Deal collision

1. choosing m dynamically .

$\alpha = \frac{n}{m}$ [load factor] n : # of <k, v>, m : size of current hst

when $\frac{n}{m} > \alpha$, increase m exponentially . If we increase m by a const, the avg running time for resize is O(n)

Dynamic Size

$m = m/2$;

$\alpha_1 > 0.8$, $m = m * 2$;

2. Hash Function

h: Int ⇒ Int , distribute evenly into [0, m)

#1. Division Method

$h(x) = x \% m$ (pick a prime num for m)

(pick another prime num when resizing)

#2. Multiplication Method

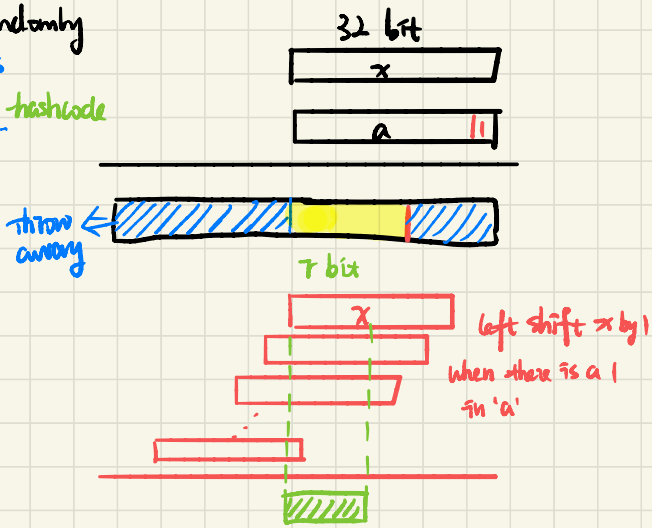
m is always 2^r (power of 2) meaning we only want r bits to rep range of m

$h(x) = (a \cdot x \% 2^{32}) >> (32 - r)$

input a is a constant choosing randomly

throw away left 32 bits

Goal: We want the most randomized hashcode based on x.



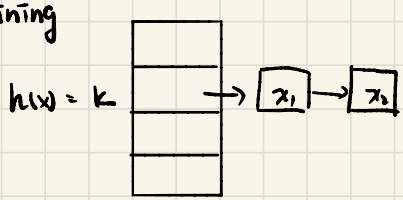
The middle r bits contains the most randomized of x. Since it contains the upper, middle and lower region info of x.

r The length of r will cover the most overlapping area.

3. Collision
- chaining
- probing

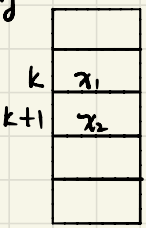
Birthday Paradox
Randomly select x people.
Probability that two people have same birthday
Minimum x such that $P(x) > 50\%$
$x = 23$

1. chaining



avg: $O(1+\alpha)$ $\alpha = \frac{n}{m}$ n items and m slots.
↓ length of LinkedList is $\frac{n}{m}$
on average.
→ $t_1 = 3$
→ $t_2 = 4$
→ $t_3 = 5$
} $t_{avg} = 4$

2. Probing



- lookup(x_1) = k
- lookup(x_2) = ? keep looking up until you find x_2

$P(\text{slot is empty}) = 1-\alpha$

lookup $O(\frac{1}{1-\alpha})$
insert $O(\frac{1}{1-\alpha})$