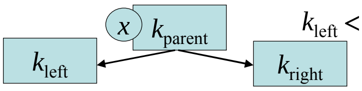
Binary Search Tree

- BST converts a static binary search into a dynamic binary search allowing to efficiently insert and delete data items
- Left-to-right ordering in a tree: for every node $\boldsymbol{x}$, the values of all the keys $\boldsymbol{k}_{\mathbf{left}}$ in the left subtree are smaller than the key $\boldsymbol{k}_{\mathbf{parent}}$ in $\boldsymbol{x}$ and the values of all the keys $\boldsymbol{k}_{\mathbf{right}}$ in the right subtree are larger than the key in $\boldsymbol{x}$:

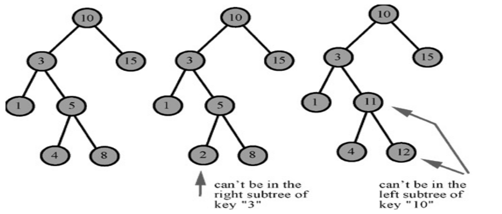


$k_{\text{left}} < k_{\text{parent}} < k_{\text{right}}$

1

Binary Search Tree

Compare the left–right ordering in a binary search tree to the bottom–up ordering in a heap where the key of each parent node is greater than or equal to the key of any child node



2

Binary Search Tree

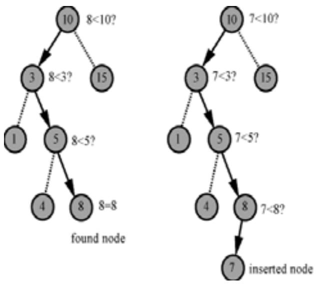
- No duplicates! (attach them all to a single item)
- Basic operations:
  - **find**: find a given search key or detect that it is not present in the tree
  - **insert**: insert a node with a given key to the tree if it is not found
  - **findMin**: find the minimum key
  - **findMax**: find the maximum key
  - **remove**: remove a node with a given key and restore the tree if necessary

3

BST: find / insert operations

find is the successful binary search

insert creates a new node at the point at which the unsuccessful search stops



4

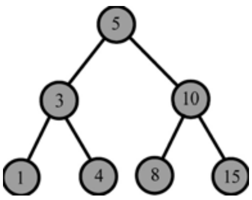
Binary Search Trees: findMin / findMax / sort

- Extremely simple: starting at the root, branch repeatedly left (**findMin**) or right (**findMax**) as long as a corresponding child exists
- The root of the tree plays a role of the pivot in quickSort
- As in QuickSort, the recursive traversal of the tree can sort the items:
  - First visit the left subtree
  - Then visit the root
  - Then visit the right subtree

5

Binary Search Tree: running time

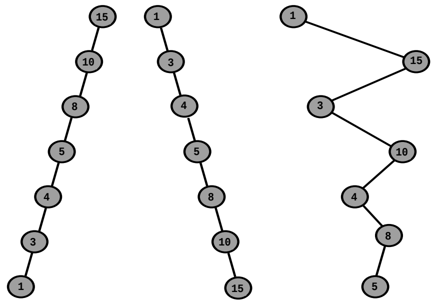
- Running time for **find**, **insert**, **findMin**, **findMax**, **sort**: $\mathrm{O}(\log n)$ average-case and $\mathrm{O}(n)$ worst-case complexity (just as in QuickSort)



BST of the depth about $\log n$

6

BST of the depth about *n*



Binary Search Tree: node removal

- **remove** is the most complex operation:
  - The removal may disconnect parts of the tree
  - The reattachment of the tree must maintain the binary search tree property
  - The reattachment should not make the tree unnecessarily deeper as the depth specifies the running time of the tree operations

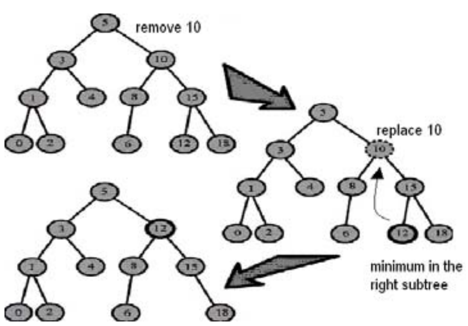
BST: how to remove a node

- If the node k to be removed is a leaf, delete it
- If the node k has only one child, remove it after linking its child to its parent node
- Thus, **removeMin** and **removeMax** are not complex because the affected nodes are either leaves or have only one child

BST: how to remove a node

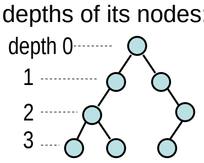
- If the node k to be removed has two children, then replace the item in this node with the item with the **smallest** key in the **right** subtree and remove this latter node from the right subtree (***Exercise:*** if possible, how can the nodes in the left subtree be used instead?)
- The second removal is very simple as the node with the smallest key does not have a left child
- The smallest node is easily found as in **findMin**

BST: an Example of Node Removal



Average-Case Performance of Binary Search Tree Operations

- **Internal path length** of a binary tree is the sum of the depths of its nodes:



IPL = 0 + 1 + 1 + 2 + 2 + 3 + 3 + 3
= 15

- **Average internal path length** $T(n)$ of the binary search trees with n nodes is $O(n \log n)$

Average-Case Performance of Binary Search Tree Operations

- If the n-node tree contains the root, the i-node left subtree, and the $(n–i–1)$-node right subtree, then:

$$\mathrm{T}(n) = n - 1 + \mathrm{T}(i) + \mathrm{T}(n–i–1)$$

because the root contributes 1 to the path length of each of the other $n - 1$ nodes

- Averaging over all i; $0 \le i < n$: the same recurrence as for QuickSort:

$$\mathrm{T}(n) = (n-1) + \tfrac{2}{n}\left(\mathrm{T}(0) + \mathrm{T}(1) + \ldots + \mathrm{T}(n-1)\right)$$

so that $\mathrm{T}(n)$ is $\mathrm{O}(n \log n)$

13

Average-Case Performance of Binary Search Tree Operations

- Therefore, the average complexity of **find** or **insert** operations is $\mathrm{T}(n)\,/\,n = \mathrm{O}(\log n)$
- For n^2 pairs of random **insert** / **remove** operations, an expected depth is $\mathrm{O}(n^{0.5})$
- In practice, for random input, all operations are about $\mathrm{O}(\log n)$ but the worst-case performance can be $\mathrm{O}(n)$!

14

Balanced Trees

- Balancing ensures that the internal path lengths are close to the optimal $n \log n$
- The average-case and the worst-case complexity is about $\mathrm{O}(\log n)$ due to their balanced structure
- But, **insert** and **remove** operations take more time on average than for the standard binary search trees
  - **AVL** tree (1962: Adelson-Velskii, Landis)
  - **Red-black** and **AA-tree**
  - **B-tree** (1972: Bayer, McCreight)

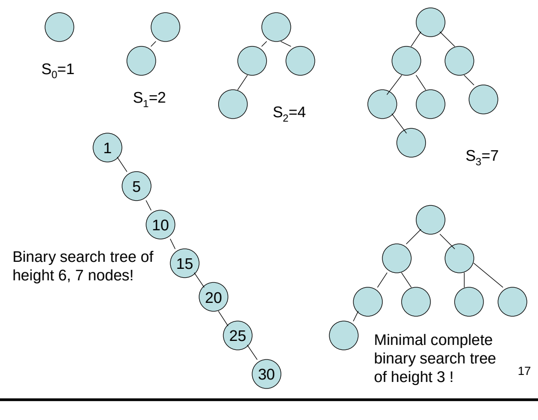
15

AVL Tree

- An AVL tree is a binary search tree with the following additional balance property:
  - for any node in the tree, the height of the left and right subtrees can differ by at most 1
  - the height of an empty subtree is -1
- The AVL-balance guarantees that the AVL tree of height h has at least c^h nodes, $c > 1$, and the maximum depth of an n-item tree is about $\log_c n$

16

Minimal AVL-trees of heights 0,1,2,3 !



17

Lemma 3.19: The height of an AVL-tree is O(log n).

Proof:

1) AVL tree of height h has less than $2^{h+1} – 1$ nodes.

2) We calculate the maximum height of an AVL tree with n nodes.

Let S_h be the size of the smallest AVL tree of the height h (it is obvious that $S_0 = 1$, $S_1 = 2$)

We can set up and solve the following recurrence relation:

$$S_h = S_{h-1} + S_{h-2} + 1,$$
$$S_h = F_{h+3} - 1$$

where F_i is the i-th Fibonacci number

18

Definition-Fibonacci number: F(n)=F(n-1)+F(n-2), F_i is ith Fibonacci number, F_1=1, F_2=1, F_3=2, F_4=F_2+F_3

$$S_h = S_{h-1} + S_{h-2} + 1,$$
$$S_h = F_{h+3} - 1$$

Proof by mathematical induction:

1) Base case: $S_0=F_3-1=1$, $S_1=F_4-1=2$

2) Assumption: $S_i=F_{i+3}-1$

3) Proof: $S_{i+1}=(F_{i+3}-1)+(F_{i+2}-1)+1=F_{i+4}-1$

19

AVL Tree

- Therefore, for each *n*-node AVL tree:

$$n \geq S_h \approx \left(\varphi^{h+3}/\sqrt{5}\right) - 1$$

where $\varphi = \left(1+\sqrt{5}\right)/2 \cong 1.618$, or

$$h \leq 1.44\log_2(n+1) - 1.328$$

- Thus, the worst-case height is at most 44% more than the minimum height of a complete binary tree

20

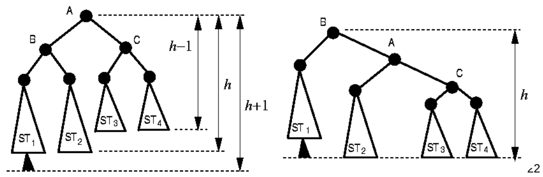
Balancing an AVLTree

- Two mirror-symmetric pairs of cases to rebalance the tree if after the insertion of a new key to the bottom the AVL property is invalidated
- Only one single or double rotation is sufficient
- Deletions are more complicated: $O(\log n)$ rotations can be required to restore the balance
- AVL balancing is not computationally efficient . Better balanced search trees: red-black, AA-trees, B-trees

21

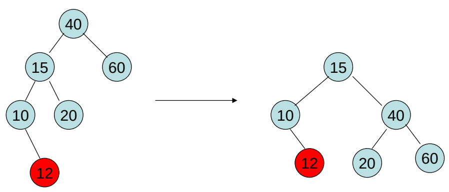
Single Rotation

- The inserted new key invalidates the AVL property
- To restore the AVL tree, the root is moved to the node B and the rest of the tree is reorganised as the BST



22

Example for inserting a new key and rebalancing the AVL-tree:



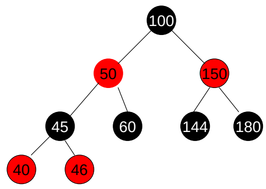
Insertion algorithm (informal):
1) Ordinary binary search tree insertion
2) rebalance the tree if necessary

23

Red-Black Tree

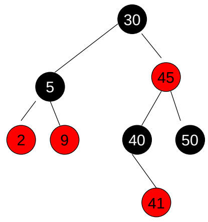
- A **red**-**black tree** is a binary search tree with the following ordering properties: Every node is coloured either **red** or **black,** the root is **black**
  - *Red Rule*: If a node is **red**, its children must be **black.**
  - *Path Rule*: Every path from the root to a leaf or to a node with one child must contain the same number of **black** nodes.

24

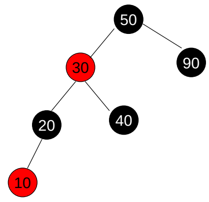


Red-Black Tree

25



A red-black tree with 8 nodes
1) RedRule is satisfied.
2) 2 black nodes in each of the 5 paths from the root to a leaf or a node with 1 child,
So the PathRule is satisfied.



A red-black tree not balanced!
A new node below 10 not possible!
This shows that red black trees are somehow balanced!

26

Red-Black Tree

If a red-black tree is complete, with all black nodes except for red leaves at the lowest level, the height of that tree will be minimal, approximately $\log_2 n$.

To get the maximum height of for a given n, we would have as many red nodes as possible on one path, and all other nodes are black. The path with all the red nodes would be about twice as long as the paths with no red elements..

This lead us to the hypothesis that the maximum height of a red-black tree is less than 2 $\log_2$n.

27

Red-Black Tree

Statement: Because every path from the root to a leaf contains *b* black nodes, there are at least 2^b -1 nodes in the tree.

- Proof (math induction):
  Base case: it is valid for $b=1$ (only the root or also 1-2 its red children)
- Let it be valid for all red-black trees with b black nodes per path
- If a tree contains $b+1$ black nodes per path and the root has 2 black children, then it contains at least
  $2\cdot(2^b-1)+1 = 2^{b+1}-1$ black nodes

28

Red-Black Tree

Proof of hypothesis that any red-black tree with height t is O(log n):

By the red rule, at most half of the nodes in the path can be red, so at least half of the nodes must be black.

That means: $b \geq h/2$

Previous proof: $n \geq 2^b - 1$

We replace b: $n \geq 2^{h/2} - 1$

We get: $h \leq 2 \cdot \log_2(n+1)$

29

Summary AVL, Red-Black trees

1) Red-black trees never get far out of balance.

2) Maximum height of an AVL-tree

$$h \leq 1.44 \log_2(n+1) - 1.328$$

Maximum height of an Red-Black tree:

$$h \leq 2 \cdot \log_2(n+1)$$

30

AA-Trees

- Software implementation of the operations insert and remove for red-black trees is a rather tricky process
- A balanced search **AA-tree** is a method of choice if deletions are needed
- The AA-tree adds one extra condition to the **red-black** tree: left children may not be red
- This condition greatly simplifies the red-black tree remove operation

31

B-Trees: Efficient External Search

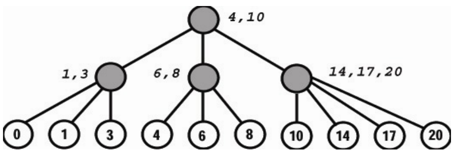
- For very big databases, even $\log_2 n$ search steps may be unacceptable
- To reduce the number of disk accesses: an optimal m-ary search tree of height about $\log_m n$

m-way branching lowers the optimal tree height by factor $\log_2 m$ (i.e., by 3.3 if m=10)

n	10^5	10^6	10^7	10^8	10^9
$\lceil \log_2 n \rceil$	17	20	24	27	30
$\lceil \log_{10} n \rceil$	5	6	7	8	9
$\lceil \log_{100} n \rceil$	3	3	4	4	5
$\lceil \log_{1000} n \rceil$	2	2	3	3	3

32

Multiway Search Tree of Order $m = 4$



- In an m-ary search tree, at most $m-1$ keys are used to decide which branch to take
- The data records are associated only with leaves, so the worst-case and average-case searches involve the tree height and the average leaf depth, respectively

33

B-Tree Definition

- A B-tree of order m is defined as an m-ary balanced tree with the following properties:
  - The root is either a leaf or it has between 2 and m children inclusive
  - Every nonleaf node (except possibly the root) has between $\lceil m/2 \rceil$ and m children inclusive

34

B-Tree Definition

- A nonleaf node with μ children has $\mu-1$ keys ($\texttt{key}_i$: i=1, …, $\mu-1$) to guide the search
- $\texttt{key}_i$ represents the smallest key in the subtree i+1
- All leaves are at the same depth
- The data items are stored at leaves, and every leaf contains between $\lceil l/2 \rceil$ and l data items, for some l that may be chosen independently of the tree order m
- Assuming that each node represents a disk block, the choice is based on the size of items that are being stored

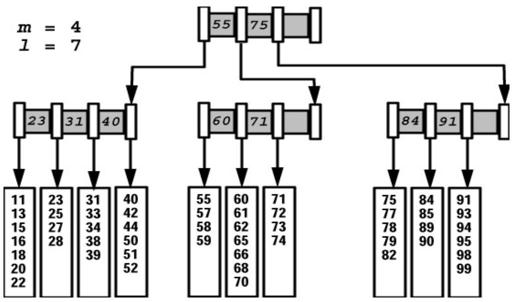
35

Naming the B-Trees

- B-trees are named after their branching factors, that is, $\lceil m/2 \rceil$ - m-tree
- A 4–7-tree is the B-tree of order $m = 7$
  - 2..7 children per root
  - 4..7 children per each non-root node
- A 2–4-tree is the B-tree of order $m = 4$
  - 2..4 children per root
  - 2..4 children per each non-root node

36

2–4-Tree



The leaf has between 2 and m children, here 3, the nonleaf nodes have between 2 and 4 children and 1 to 3 keys, the leafs have a maximum of 7 records

Example of choosing m, l

- Let one disk block holds 8192 bytes.
- Each key uses 32 bytes.
- The branch is a number of another disk block, so let a branch be 4 bytes.
- B-tree of order m: $m-1$ keys per node plus m branches
- So the largest order m that 1 node fits in 1 disk block:

  $32(m-1)+4m = 36m-32 < 8192 \rightarrow m=228$
- Because the block size is 8192 bytes, $l = 32$ records of size 256 bytes fit in a single block
- Let there be 10^7 data records, 256 bytes per record

Example of choosing m, l

- Each leaf has between 16..32 data records inclusive, and each internal node, except from the root, branches in 114 – 228 ways
- 10^7 records can be stored in 312500 – 625000 leaves (= 10^7 / (16 .. 32))
- In the worst case the leaves would be on the level 4 ($114^2 = 12996 < 625{,}000 < 114^3 = 1481544$)

Analysis of B-Trees

- A search or an insertion in a B-tree of order m with n data items requires fewer than $\lceil \log_m n \rceil$ disk accesses
- In practice, $\lceil \log_m n \rceil$ is almost constant as long as m is not small
- Data insertion is simple until the corresponding leaf is not already full; then it must be split into 2 leaves, and the parent(s) should be updated

Analysis of B-Trees

- Additional disk writes for data insertion and deletion are extremely rare
- An algorithm analysis beyond the scope of this course shows that both insertions, deletions, and retrievals of data have only $\log_{m/2} n$ disk accesses in the worst case (e.g., $\lceil \log_{114} 625000 \rceil = 3$ in the above example)

Symbol Table and Hashing

- A (symbol) table is a set of table entries, (K,V)
- Each entry contains:
  - a unique key, K, and
  - a value (information), V
- Each key uniquely identifies its entry

Key K		Associated value V		
Code	k	*City*	*Country*	*State*
AKL	271	Auckland	NZ	
DCA	2080	Washington	USA	DC
FRA	3822	Frankfurt	Germany	
GLA	4342	Glasgow	UK	Scotland
HKG	4998	Hong Kong	China	
LAX	7459	Los Angeles	USA	California

$k=26^2c_0+26c_1+c_2$, c_0, c_1, c_2 are the integer codes for the English alphabet, 0-A, 1-B, 2-C, 3…….

A table is a mapping of keys to values.

43

Example:

We have a table with 1000 elements. Each element has a security number as a key. We need to transform the key into an index in our array. To allow fast access, we can take the right most 3 digits of the number.

For example: 214303261 would be stored at index 261 or 033518000 would be stored at 0.

You can see two distinct keys can have the same 3 digits at the end. -collision!

Hashing is the process of transforming a key into a table index.
Hash-function: performs an easily computable operation on the key and returns the **hash value**.

44

Symbol Table and Hashing

- Once the entry (K,V) is found, its value V, may be updated, it may be retrieved, or the entire entry, (K,V) , may be removed from the table
- If no entry with key K exists in the table, a new table entry having K as its key may be inserted in the table
- Hashing is a technique of storing values in the tables and searching for them in linear, $O(n)$, worst-case and extremely fast, $O(1)$, average-case time

45

Basic Features of Hashing

- Hashing computes an integer, called the hash code, for each object (key)
- The computation, called the hash function, $h(K)$, maps objects (e.g., keys K) to the array indices (e.g., $0, 1, \ldots, i_{max}$)
- An object having a key value K should be stored at location $h(K)$, and the hash function must always return a valid index for the array

46

Basic Features of Hashing

- A perfect hash function produces a different index value for every key. But such a function cannot be always found.
- **Collision:** if two distinct keys, $K_1 \neq K_2$, map to the same table address, $h(K_1) = h(K_2)$
- **Collision resolution policy:** how to find additional storage in which to store one of the collided table entries

47

How Common Are Collisions?

- **Von Mises Birthday Paradox:**

  if there are more than 23 people in a room, the chance is greater than 50% that two or more of them will have the same birthday
- Thus, in the table that is only 6.3% full (since $23/365 = 0.063$) there is a “good” chance of a collision!

48

How Common Are Collisions?

- **Probability of no collision** (that is, that none of the n items collides, being randomly tossed into a table of size N):

$$Q_N(1) = 1 \equiv \frac{N}{N}; \quad Q_N(2) = Q_N(1)\frac{N-1}{N} \equiv \frac{N(N-1)}{N^2};$$

$$Q_N(3) = Q_N(2)\frac{N-2}{N} \equiv \frac{N(N-1)(N-2)}{N^3}; \quad \ldots$$

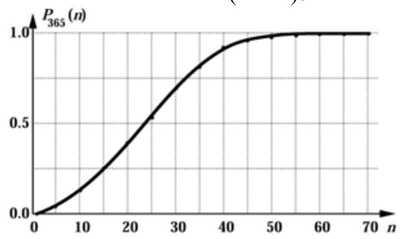
$$Q_N(n) = Q_N(n-1)\frac{N-n+1}{N} \equiv \frac{N(N-1)\ldots(N-n+1)}{N^n}$$

49

Probability $P_N(n)$ of One or More Collisions

$$P_N(n) = 1 - Q_N(n) = 1 - \frac{N!}{N^n(N-n)!}$$

n	%	$P_{365}(n)$
10	2.7	0.1169
20	5.5	0.4114
30	8.2	0.7063
40	11.0	0.8912
50	13.7	0.9704
60	16.4	0.9941



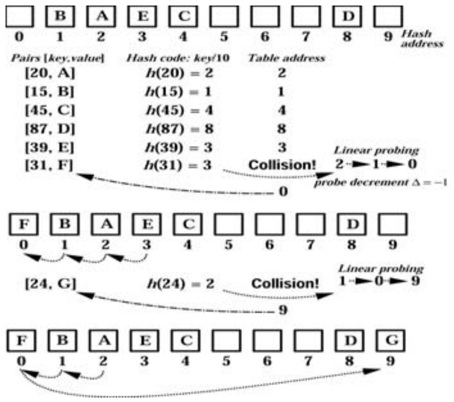
50

Open Addressing with Linear Probing (OALP)

- The simplest collision resolution policy:
  - to successively search for the first empty entry at a lower location
  - if no such entry, then ``wrap around'' the table
- **Drawbacks:** clustering of keys in the table

51

OALP example:
$N = 10$
h(k)=int(k/10)



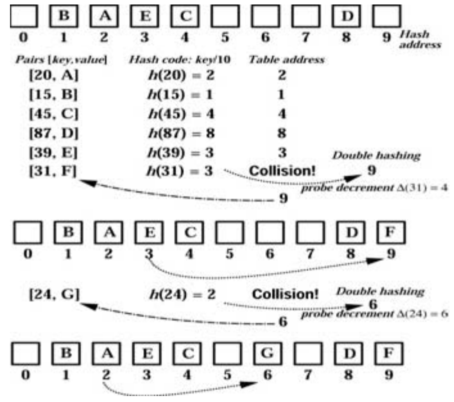
52

Open Addressing with Double Hashing (OADH)

- Better collision resolution policy reducing the likelihood of clustering:
  - to hash the collided key again using a different hash function and
  - to use the result of the second hashing as an increment for probing table locations (including wraparound)

53

OADH
$N = 10$,
h(k)=int(h/10)
d(k)=h(k)+k mod 10



54

Two More Collision Resolution Techniques

- Open addressing has a problem when significant number of items need to be deleted as logically deleted items must remain in the table until the table can be reorganised
- Two techniques to attenuate this drawback:
  - Chaining
  - Hash bucket

Chaining and Hash Bucket

- **Chaining:** all keys collided at a single hash address are placed on a linked list, or chain, started at that address
- **Hash bucket:** a big hash table is divided into a number of small sub-tables, or buckets
  - the hush function maps a key into one of the buckets
  - the keys are stored in each bucket sequentially in increasing order

Choosing a hash function

- Four basic methods: division, folding, middle-squaring, and truncation
- **Division:**
  - choose a prime number as the table size N
  - convert keys, K, into integers
  - use the remainder $\mathrm{h}(K) = K \bmod N$ as a hash value of the key K
  - find a double hashing decrement using the quotient,

$$\Delta \mathrm{K} = \max\{1, (K/N) \bmod N\}$$

Choosing a hash function

- **Folding:**
  - divide the integer key, K, into sections
  - add, subtract, and/or multiply them together for combining into the final value, $\mathrm{h}(K)$
- Example:

  K=013402122 → sections 013, 402, 122 → $\mathrm{h}(K) = 013 + 402 + 122 = 537$

Choosing a hash function

- **Middle-squaring:**
  - choose a middle section of the integer key, K
  - square the chosen section
  - use a middle section of the result as $\mathrm{h}(K)$
- Example:

  $K = 013402122$ → middle: 402 → 402^2=161404 → middle: $\mathrm{h}(K) = 6140$

Choosing a hash function

- **Truncation:**
  - delete part of the key, K
  - use the remaining digits (bits, characters) as $\mathrm{h}(K)$
- Example:

  K=013402122 → last 3 digits: $\mathrm{h}(K) = 122$
- Notice that truncation does not spread keys uniformly into the table; thus it is often used in conjunction with other methods

Universal Class by Division

- **Theorem** (universal class of hash functions by division):
  - Let the size of a key set, $\mathbf{K}$, be a prime number:

    $$|\mathbf{K}| = M$$

  - Let the members of $\mathbf{K}$ be regarded as the integers $\{0,\ldots,M-1\}$
  - For any numbers $a \in \{1,\ldots,M\text{-}1\}$; $b \in \{0,\ldots,M-1\}$ let

    $$\mathrm{h}_{a,b}(k) = \big((a \cdot k + b) \bmod M\big) \bmod N$$

61

Table ADT Representations: Comparative Performance

Operation	Representation		
	Sorted array	AVL tree	Hash table
Initialize	$O(N)$	$O(1)$	$O(N)$
Is full?	$O(1)$	$O(1)$	$O(1)$
Search*)	$O(\log N)$	$O(\log N)$	$O(1)$
Insert	$O(N)$	$O(\log N)$	$O(1)$
Delete	$O(N)$	$O(\log N)$	$O(1)$
Enumerate	$O(N)$	$O(N)$	$O(N \log N)$**)

*) also: Retrieve, Update **)To enumerate a hash table, entries must first be sorted in ascending order of keys that takes $O(N \log N)$ time 62