CS143: Index

Book Chapters:
(4^{th}) 12.1-3, 12.5-8
(5^{th}) 12.1-3, 12.6-8, 12.10

Topics to Learn

- Important concepts
  - Dense index vs. sparse index
  - Primary index vs. secondary index
    (= clustering index vs. non-clustering index)
  - Tree-based vs. hash-based index
- Tree-based index
  - Indexed sequential file
  - B+-tree
- Hash-based index
  - Static hashing
  - Extendible hashing

Basic Problem

- SELECT *
  FROM Student
  WHERE sid = 40

sid	name	GPA
20	Elaine	3.2
70	Peter	2.6
40	Susan	3.7

- How can we answer the query?

Random-Order File

- How do we find sid=40?

sid	name	GPA
20	Susan	3.5
60	James	1.7
70	Peter	2.6
40	Elaine	3.9
30	Christy	2.9

Sequential File

- Table sequenced by sid. Find sid=40?

sid	name	GPA
20	Susan	3.5
30	James	1.7
40	Peter	2.6
50	Elaine	3.9
60	Christy	2.9

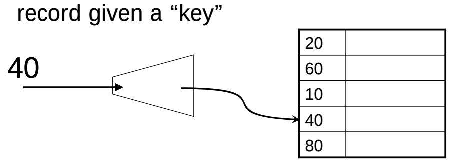
Binary Search

- 100,000 records
- Q: How many blocks to read?

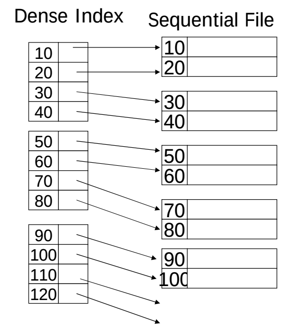
- Any better way?
  - In a library, how do we find a book?

Basic Idea

- Build an “index” on the table
  - An auxiliary structure to help us locate a record given a “key”



Dense, Primary Index

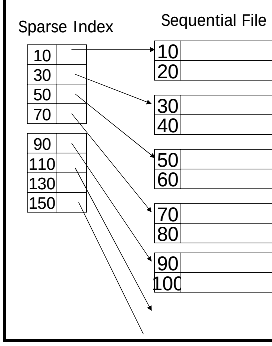


- Primary index (clustering index)
  - Index on the search key
- Dense index
  - (key, pointer) pair for every record
- Find the key from index and follow pointer
  - Maybe through binary search
- Q: Why dense index?
  - Isn’t binary search on the file the same?

Why Dense Index?

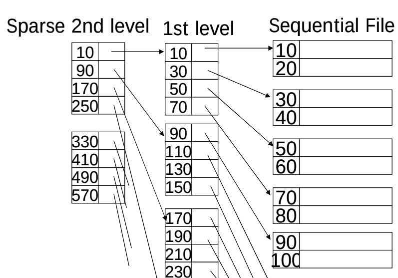
- Example
  - 10,000,000 records (900-bytes/rec)
  - 4-byte search key, 4-byte pointer
  - 4096-byte block. Unspanned tuples
- Q: How many blocks for table (how big)?
- Q: How many blocks for index (how big)?

Sparse, Primary Index



- Sparse index
  - (key, pointer) pair per every “block”
  - (key, pointer) pair points to the first record in the block
- Q: How can we find 60?

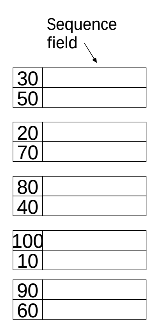
Multi-level index



Q: Why multi-level index?

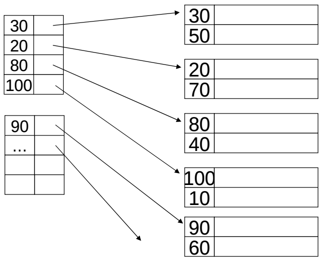
Q: Does dense, 2nd level index make sense?

Secondary (non-clustering) Index

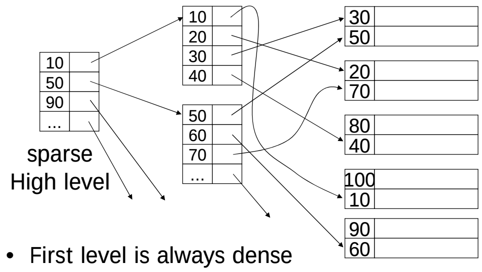


- Secondary (non-clustering) index
  - When tuples in the table are not ordered by the index search key
    - Index on a non-search-key for sequential file
    - Unordered file
- Q: What index?
  - Does sparse index make sense?

Sparse and secondary index?



Secondary index



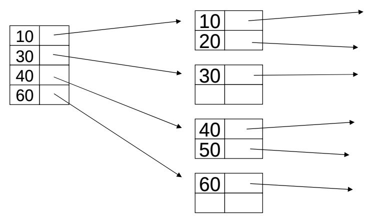
- First level is always dense
- Sparse from the second level

Important terms

- Dense index vs. sparse index
- Primary index vs. secondary index
  - Clustering index vs. non-clustering index
- Multi-level index
- Indexed sequential file
  - Sometimes called ISAM (indexed sequential access method)
- Search key (≠ primary key)

Insertion

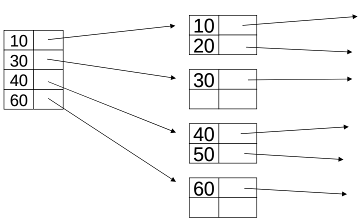
Insert 35



Q: Do we need to update higher-level index?

Insertion

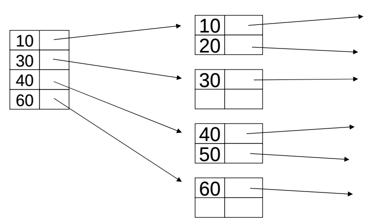
Insert 15 (overflow)



Q: Do we need to update higher-level index?

Insertion

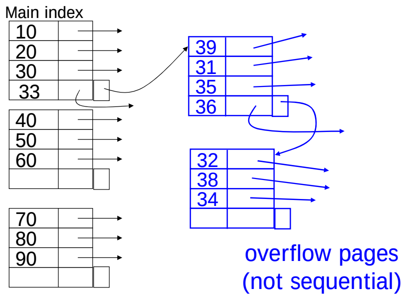
Insert 15 (redistribute)



Q: Do we need to update higher-level index?

Potential performance problem

After many insertions…



19

Traditional Index (ISAM)

- Advantage
  - Simple
  - Sequential blocks
- Disadvantage
  - Not suitable for updates
  - Becomes ugly (loses sequentiality and balance) over time

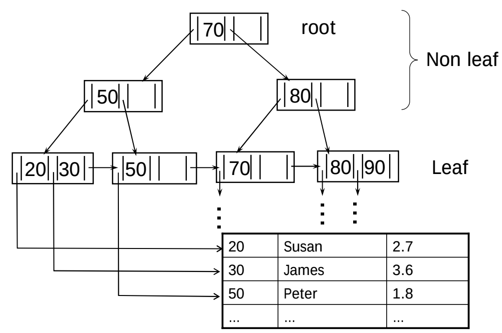
20

B+Tree

- Most popular index structure in RDBMS
- Advantage
  - Suitable for dynamic updates
  - Balanced
  - Minimum space usage guarantee
- Disadvantage
  - Non-sequential index blocks

21

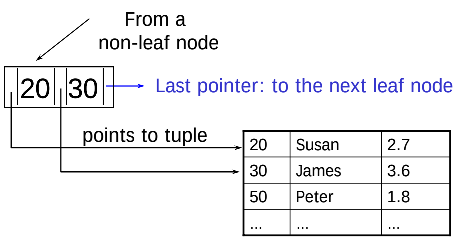
B+Tree Example (n=3)



Balanced: All leaf nodes are at the same level

22

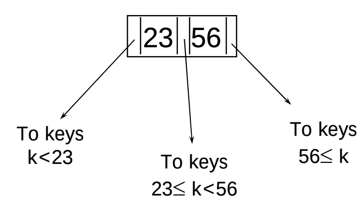
Sample Leaf Node (n=3)



- n: max # of pointers in a node
- All pointers (except the last one) point to tuples
- At least half of the pointers are used.
  (more precisely, $\lceil (n+1)/2 \rceil$ pointers)

23

Sample Non-leaf Node (n=3)

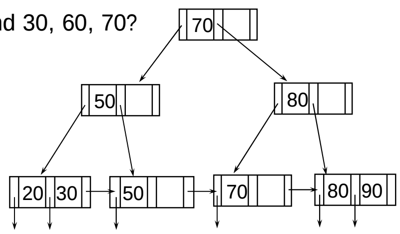


- Points to the nodes one-level below
  - No direct pointers to tuples
- At least half of the ptrs used (precisely, $\lceil n/2 \rceil$)
  - except root, where at least 2 ptrs used

24

Search on B+tree

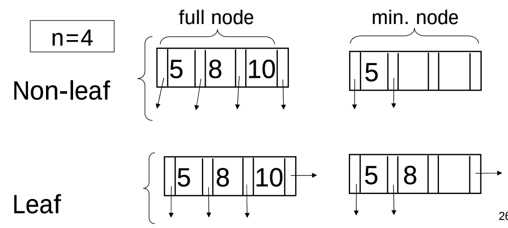
- Find 30, 60, 70?



- Find a greater key and follow the link on the left
  (Algorithm: Figure 12.10 on textbook)

Nodes are never too empty

- Use at least
  Non-leaf: $\lceil n/2 \rceil$ pointers
  Leaf: $\lceil (n+1)/2 \rceil$ pointers



Number of Ptrs/Keys for B+tree

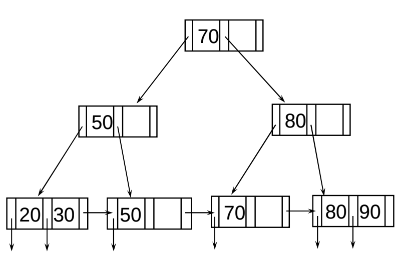
	Max Ptrs	Max keys	Min ptrs	Min keys
Non-leaf (non-root)	n	n-1	$\lceil n/2 \rceil$	$\lceil n/2 \rceil$-1
Leaf (non-root)	n	n-1	$\lceil (n+1)/2 \rceil$	$\lceil (n-1)/2 \rceil$
Root	n	n-1	2	1

B+Tree Insertion

(a) simple case (no overflow)
(b) leaf overflow
(c) non-leaf overflow
(d) new root

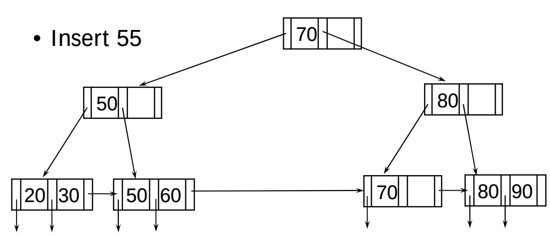
Insertion (Simple Case)

- Insert 60



Insertion (Leaf Overflow)

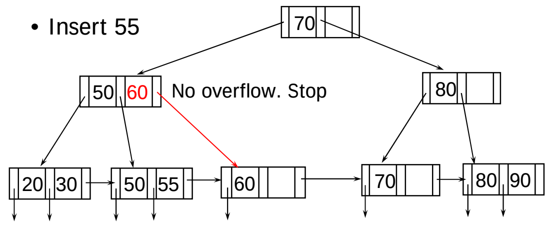
- Insert 55



- No space to store 55

Insertion (Leaf Overflow)

- Insert 55

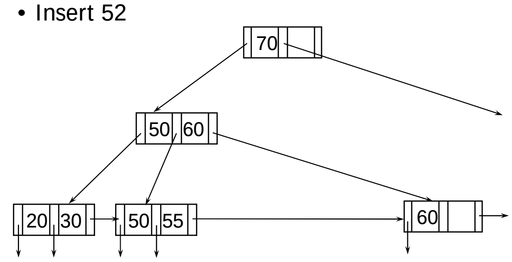


- Q: After split, leaf nodes always half full?

31

Insertion (Non-leaf Overflow)

- Insert 52

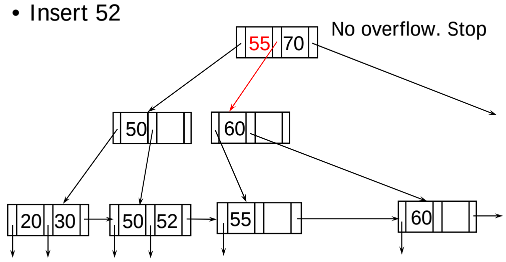


Leaf overflow. Split and copy the first key of the new node

32

Insertion (Non-leaf Overflow)

- Insert 52

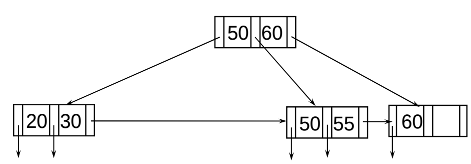


Q: After split, non-leaf at least half full?

33

Insertion (New Root Node)

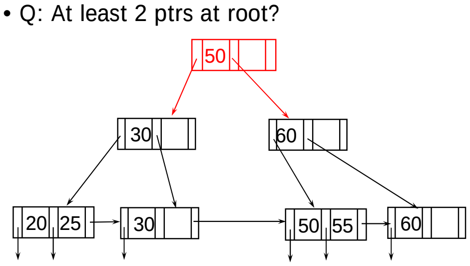
- Insert 25



34

Insertion (New Root Node)

- Insert 25
- Q: At least 2 ptrs at root?



35

B+Tree Insertion

- Leaf node overflow
  - The first key of the new node is *copied* to the parent
- Non-leaf node overflow
  - The middle key is *moved* to the parent
- Detailed algorithm: Figure 12.13

36

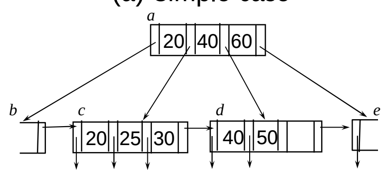
B+Tree Deletion

(a) Simple case (no underflow)
(b) Leaf node, coalesce with neighbor
(c) Leaf node, redistribute with neighbor
(d) Non-leaf node, coalesce with neighbor
(e) Non-leaf node, redistribute with neighbor

In the examples, n = 4
- Underflow for non-leaf when fewer than $\lceil n/2 \rceil$ = 2 ptrs
- Underflow for leaf when fewer than $\lceil (n+1)/2 \rceil$ = 3 ptrs
- Nodes are labeled as $a, b, c, d, \ldots$

37

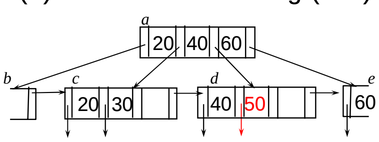
(a) Simple case



- Delete 25

38

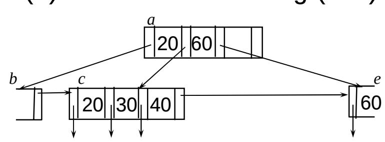
(b) Coalesce with sibling (leaf)



- Delete 50

39

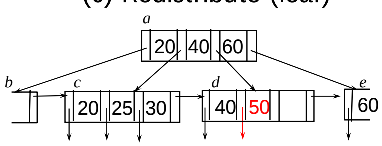
(b) Coalesce with sibling (leaf)



- Delete 50
  - Check underflow at a. Min 2 ptrs, currently 3

40

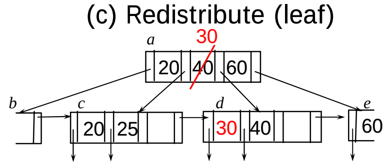
(c) Redistribute (leaf)



- Delete 50

41

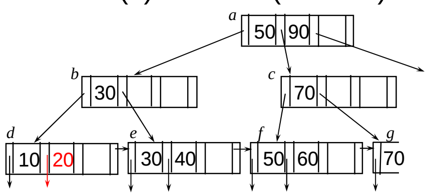
(c) Redistribute (leaf)



- Delete 50
  - No underflow at a. Done.

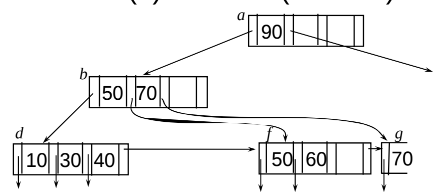
42

(d) Coalesce (non-leaf)



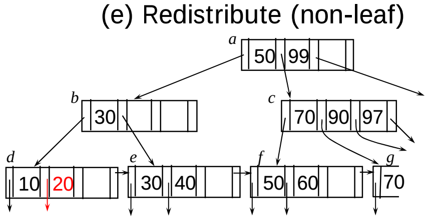
- Delete 20
  - Underflow! Merge *d* with *e*.
    - Move everything in the right to the left

(d) Coalesce (non-leaf)



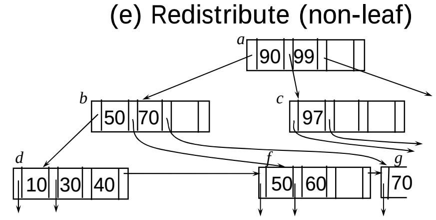
- Delete 20
  - Underflow at *a*? Min 2 ptrs. Currently 2. Done.

(e) Redistribute (non-leaf)



- Delete 20
  - Underflow! Merge *d* with *e*.

(e) Redistribute (non-leaf)



- Delete 20
  - Underflow at *a*? Min 2 ptrs, currently 3. Done

Important Points

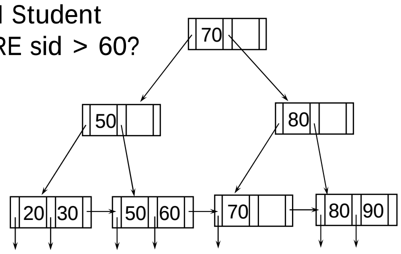
- Remember:
  - For ***leaf node*** merging, we ***delete*** the mid-key from the parent
  - For ***non-leaf node*** merging/redistribution, we ***pull down*** the mid-key from their parent.
- Exact algorithm: Figure 12.17
- In practice
  - Coalescing is often not implemented
    - Too hard and not worth it

Where does *n* come from?

- *n* determined by
  - Size of a node
  - Size of search key
  - Size of an index pointer
- Q: 1024B node, 10B key, 8B ptr → *n*?

Question on B+tree

- SELECT *
  FROM Student
  WHERE sid > 60?



Summary on tree index

- Issues to consider
  - Sparse vs. dense
  - Primary (clustering) vs. secondary (non-clustering)
- Indexed sequential file (ISAM)
  - Simple algorithm. Sequential blocks
  - Not suitable for dynamic environment
- B+trees
  - Balanced, minimum space guarantee
  - Insertion, deletion algorithms

Index Creation in SQL

- `CREATE INDEX <indexname> ON <table>(<attr>,<attr>,…)`
- Example
  - `CREATE INDEX stidx ON Student(sid)`
    - Creates a B+tree on the attributes
    - Speeds up lookup on sid

Primary (Clustering) Index

- MySQL:
  - Primary key becomes the clustering index
- DB2:
  - **`CREATE INDEX idx ON Student(sid)`** **<u>CLUSTER</u>**
  - Tuples in the table are sequenced by sid
- Oracle: Index-Organized Table (IOT)
  - **`CREATE TABLE T (`**
    `...`
    `)` **<u>ORGANIZATION INDEX</u>**
  - B+tree on primary key
  - Tuples are stored at the leaf nodes of B+tree
- Periodic reorganization may still be necessary to improve range scan performance

Next topic

- Hash index
  - Static hashing
  - Extendible hashing

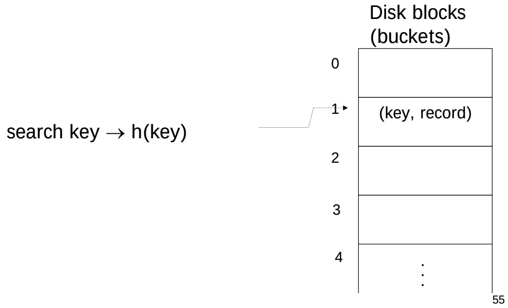
What is a Hash Table?

- Hash Table
  - Hash function
    - $h(k)$: key → integer [0...n]
    - e.g., h('Susan') = 7
  - Array for keys: T[0...n]
  - Given a key k, store it in T[$h(k)$]

h(Susan) = 4
h(James) = 3
h(Neil) = 1

0	
1	Neil
2	
3	James
4	Susan
5	

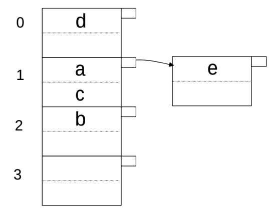
Hashing for DBMS (Static Hashing)



55

Overflow and Chaining

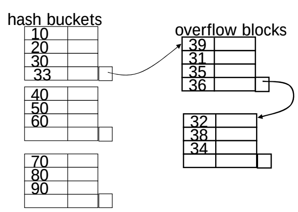
- Insert
  h(a) = 1
  h(b) = 2
  h(c) = 1
  h(d) = 0
  h(e) = 1
- Delete
  h(b) = 2
  h(c) = 1



56

Major Problem of Static Hashing

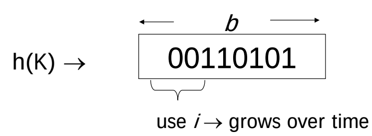
- How to cope with growth?
  – Data tends to grow in size
  – Overflow blocks unavoidable



57

Extendible Hashing (two ideas)

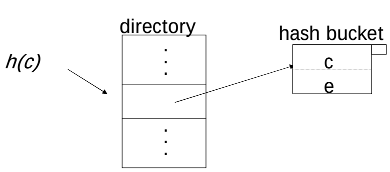
(a) Use *i* of *b* bits output by hash function



58

Extendible Hashing (two ideas)

(b) Use directory that maintains pointers to hash buckets (indirection)

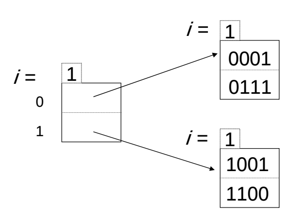


59

Example

- h(k) is 4 bits; 2 keys/bucket

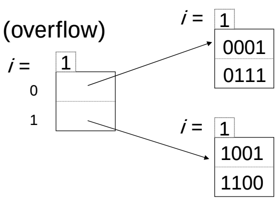
Insert 0111



60

Example

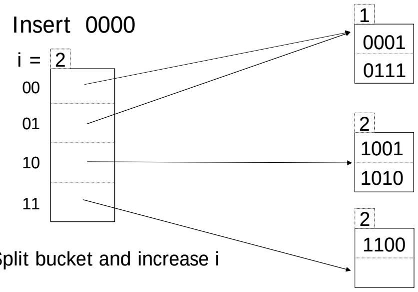
Insert 1010 (overflow)



Increase i of the bucket. Split it.

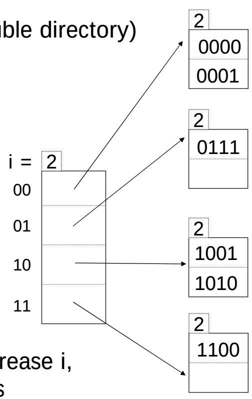
Example

Insert 0000



Split bucket and increase i

Insert 0011 (double directory)

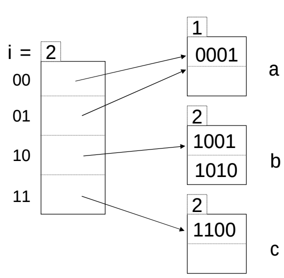


Split bucket, increase i,
redistribute keys

Extendible Hashing: Deletion

- Two options
  a) No merging of buckets
  b) Merge buckets and shrink directory if possible

Delete 1010 (merge buckets and shrink directory)



Bucket Merge Condition

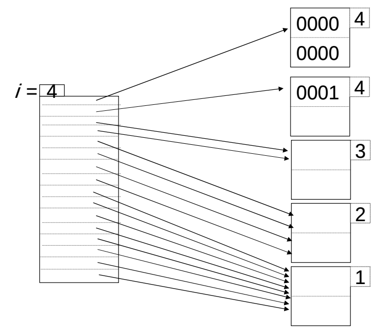
- Bucket merge condition
  - Bucket i's are the same
  - First (i-1) bits of the hash key are the same
- Directory shrink condition
  - All bucket i's are smaller than the directory i

Questions on Extendible Hashing

- Can we provide minimum space guarantee?

67

Space Waste



68

Hash index summary

- Static hashing
  - – Overflow and chaining
- Extendible hashing
  - – Can handle growing files
    - No periodic reorganizations
  - – Indirection
    - Up to 2 disk accesses to access a key
  - – Directory doubles in size
    - Not too bad if the data is not too large

69

Hashing vs. Tree

- Can an extendible-hash index support?

```
SELECT
FROM R
WHERE R.A > 5
```

- Which one is better, B+tree or Extendible hashing?

```
SELECT
FROM R
WHERE R.A = 5
```

70