Recitation 4

B Trees

Question

What advantage does balanced BST provide over storing data in an unsorted array?

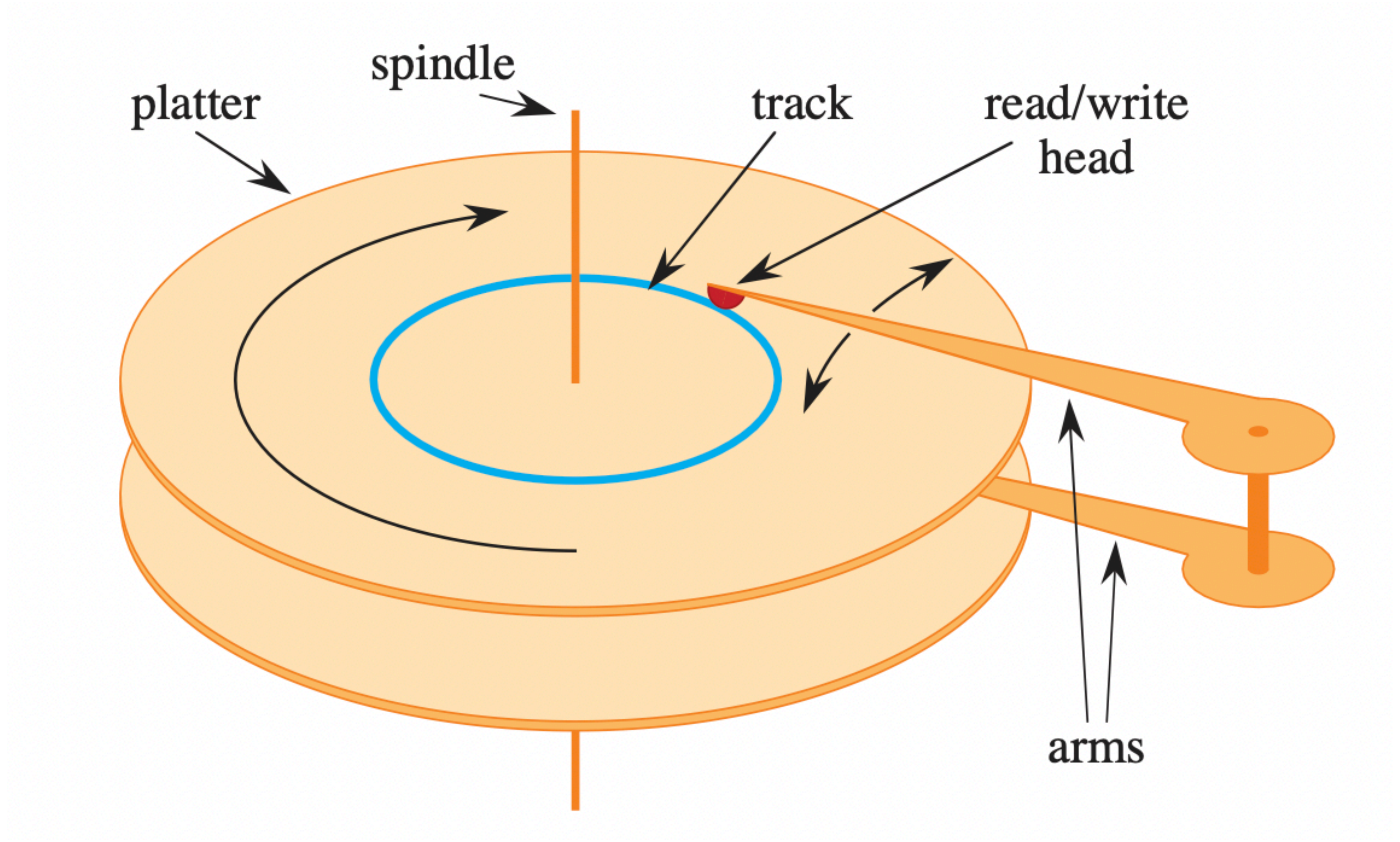
Question

What advantage does balanced BST provide over storing data in an unsorted array?

Access time.

As you move forward with the course you will notice computer scientists really care a lot about access time. In your lectures after trees you are going to learn about hash tables which has O(1) access time on average.

Access time in Memory



Typical Image of a Hard Disk Drive (HDD)

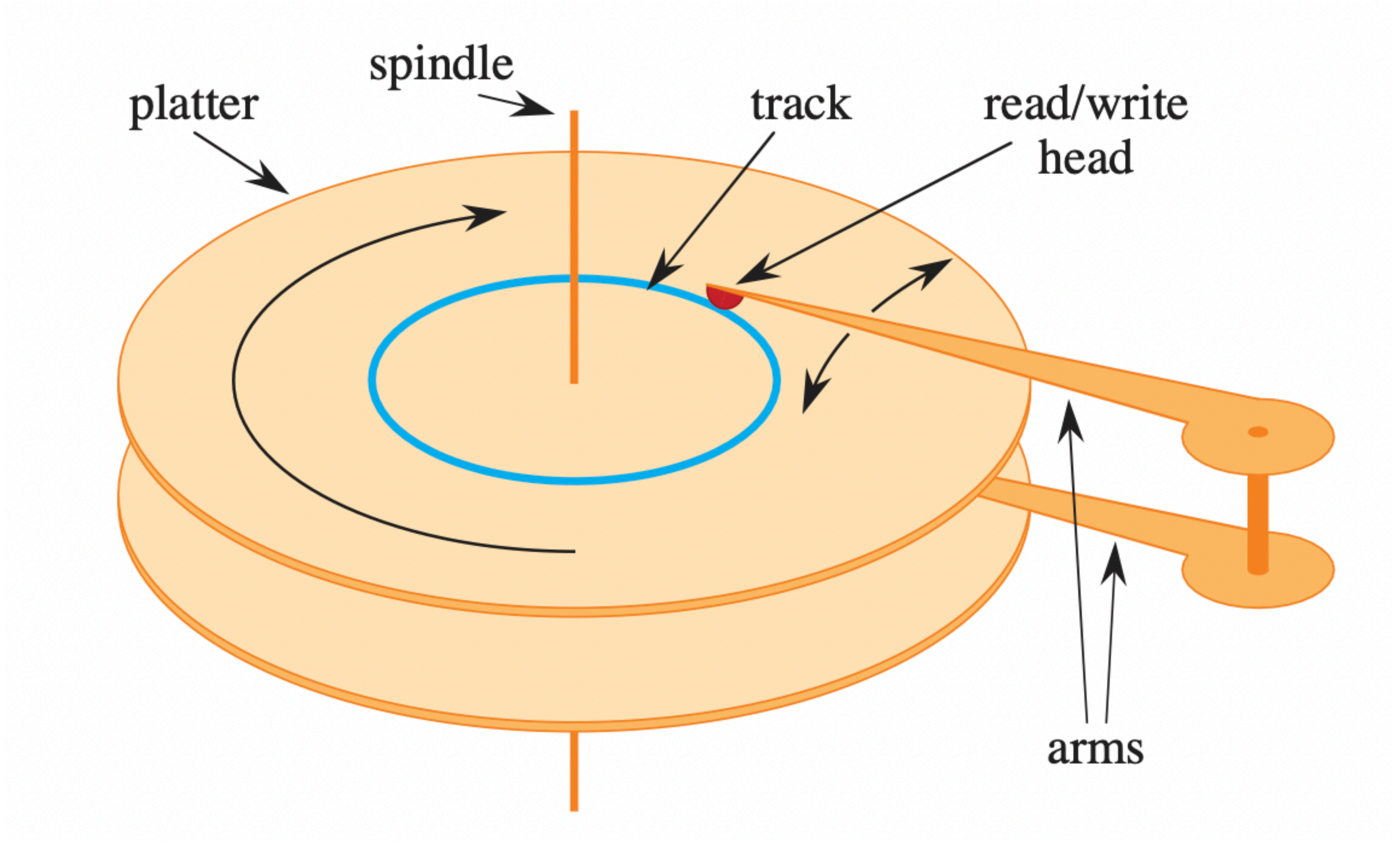
Data are stored in the track.

To access a data from the HDD, the platter rotates towards the read/write head. So, that the read/write head, (which can only move towards and backwards from the spindle) can read the data stored in the track when it comes under the read write head.

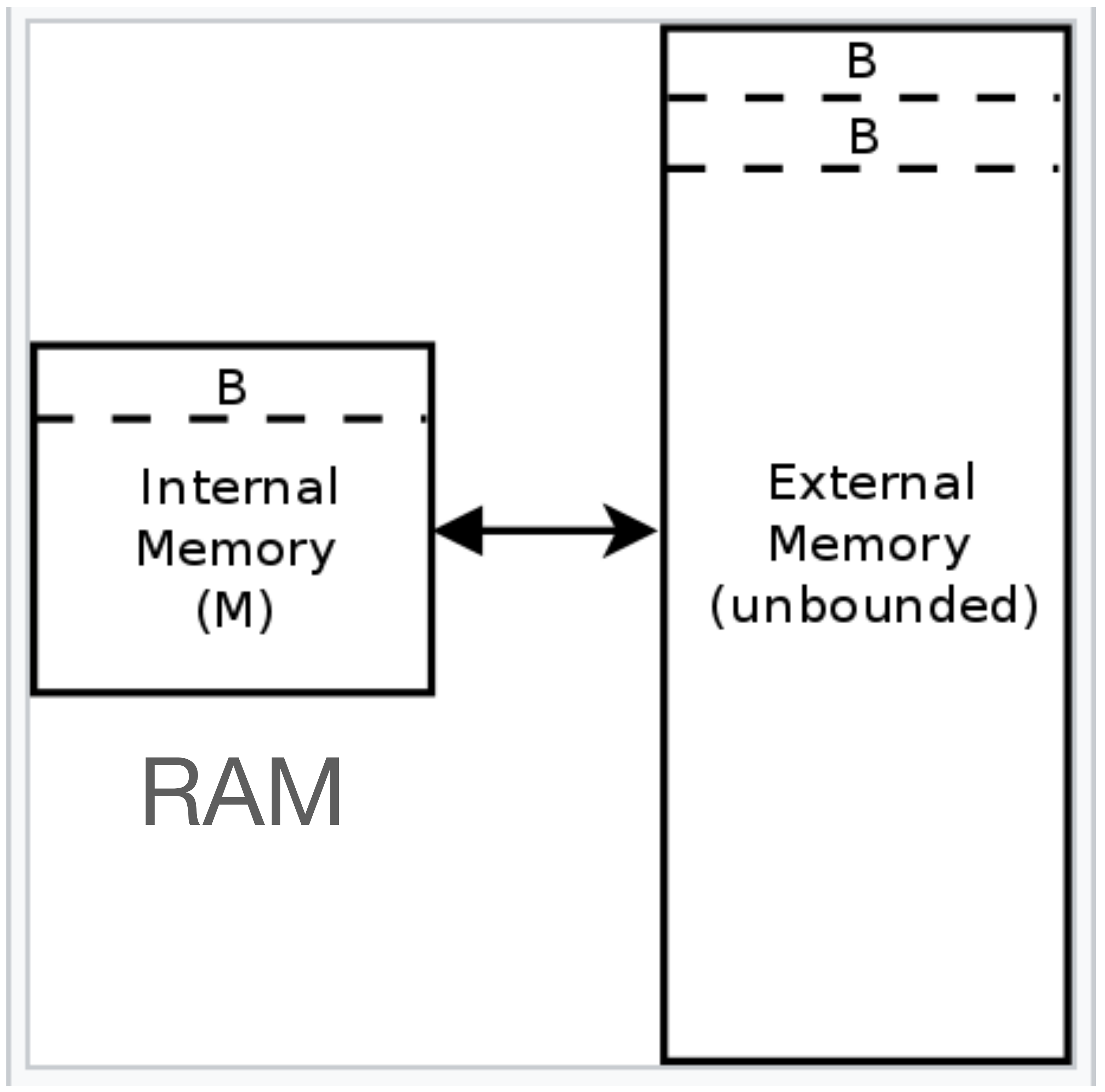
This takes 4ms time on average to access data

To make up for this latency the read/write head reads a lot of data at once (i.e. it reads the data in blocks)

Access time in Memory

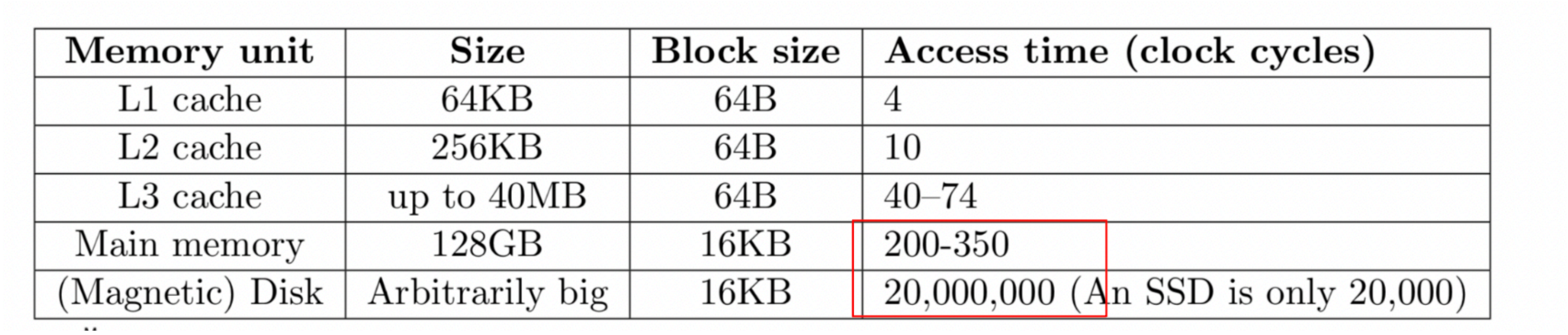


Typical Image of a Hard Disk Drive



Secondary storage

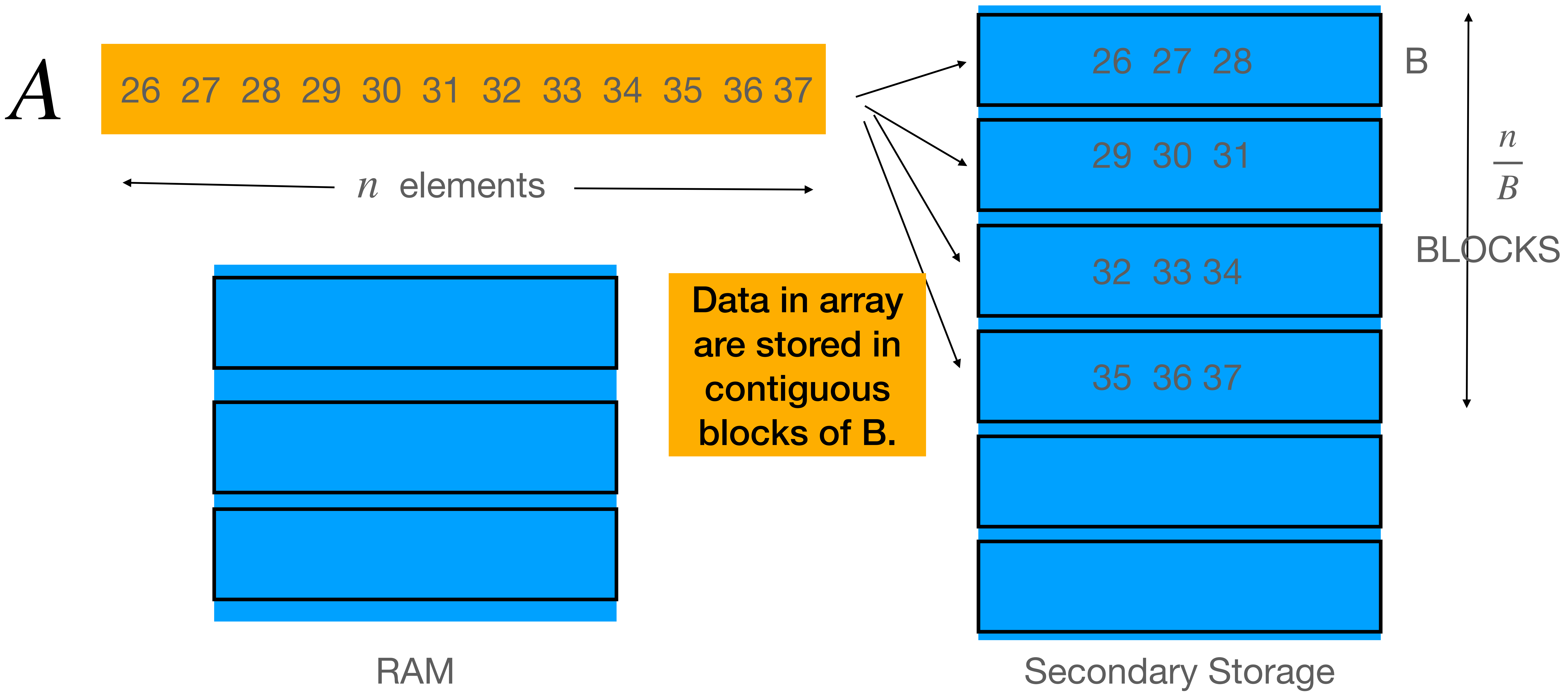
Memory Hierarchy Structure



Memory unit	Size	Block size	Access time (clock cycles)
L1 cache	64KB	64B	4
L2 cache	256KB	64B	10
L3 cache	up to 40MB	64B	40–74
Main memory	128GB	16KB	200-350
(Magnetic) Disk	Arbitrarily big	16KB	20,000,000 (An SSD is only 20,000)

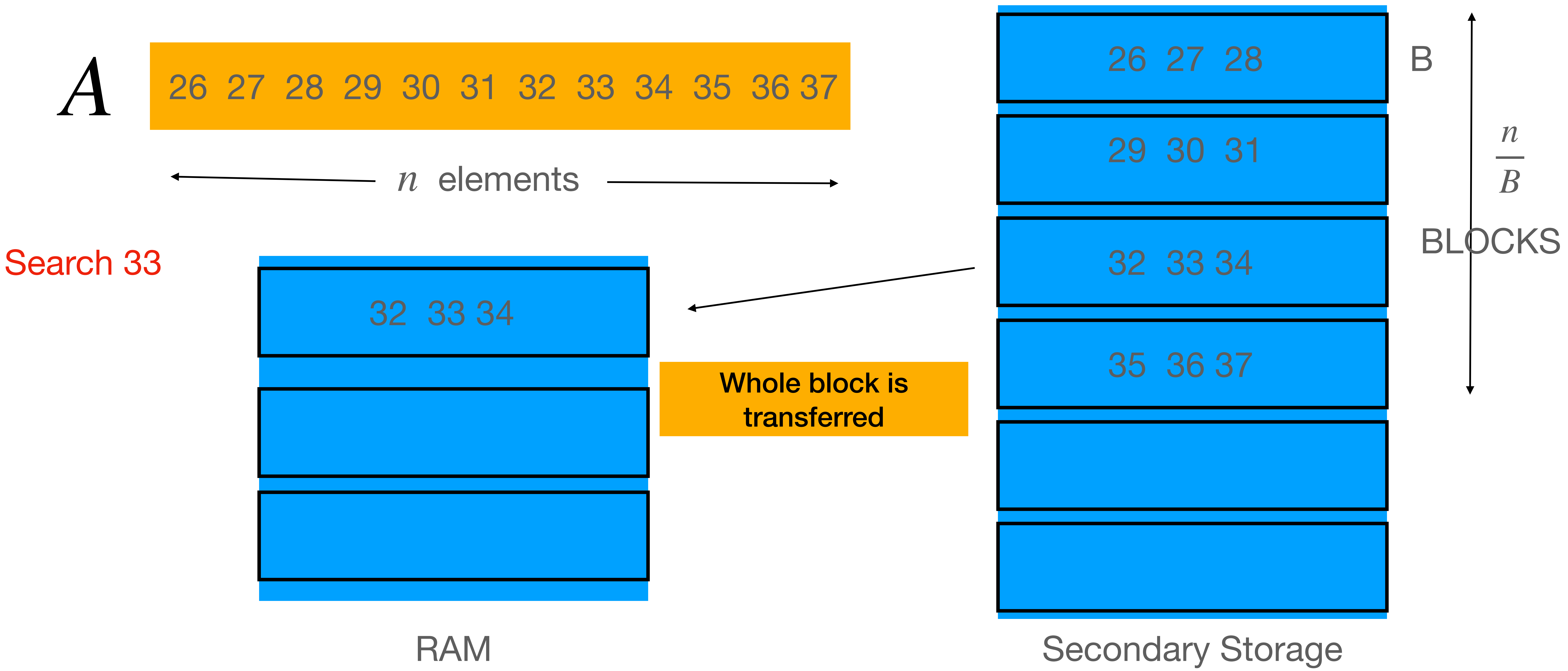
Efficient way to store data

In practice, searching a data in an array is much faster than searching a data in a linked list, only because of efficient way of storing data.

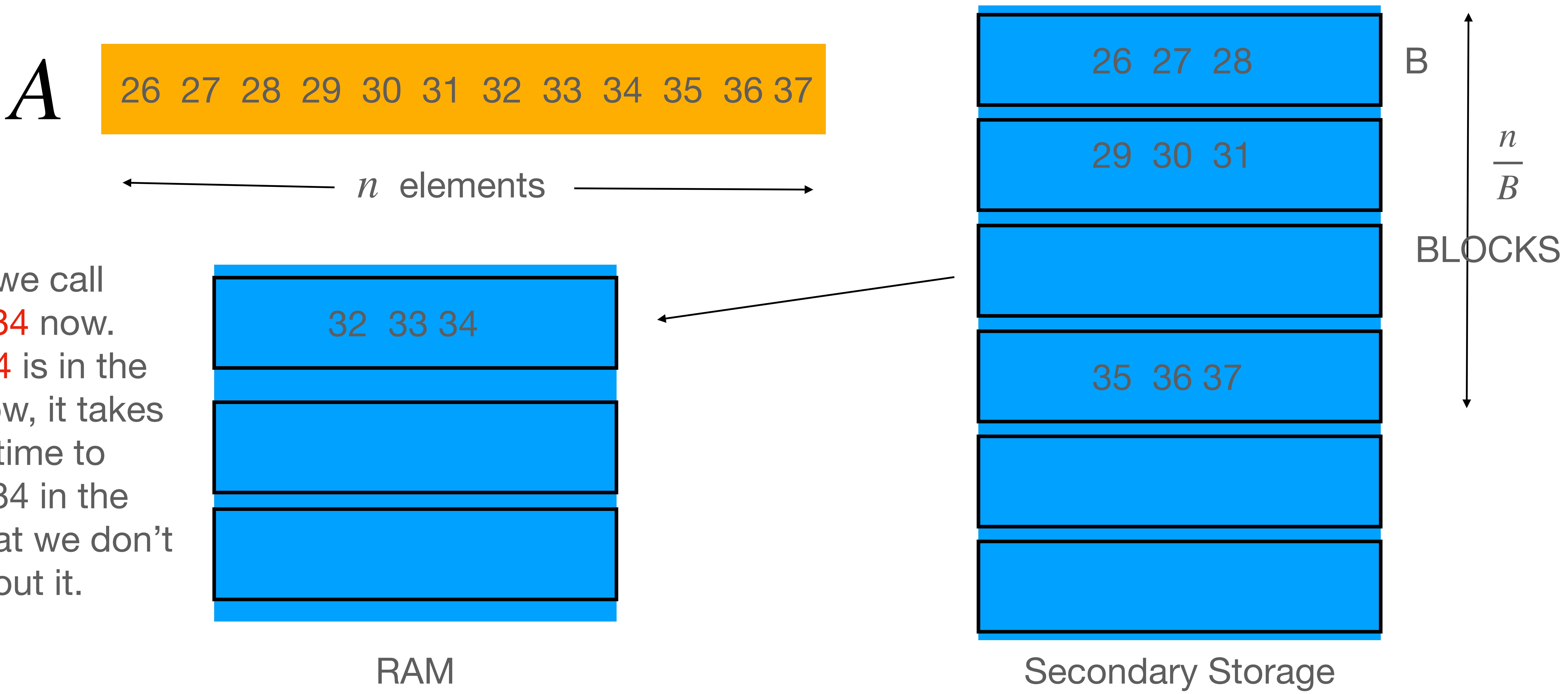


Data in array are stored in contiguous blocks of B.

Efficient way to store data

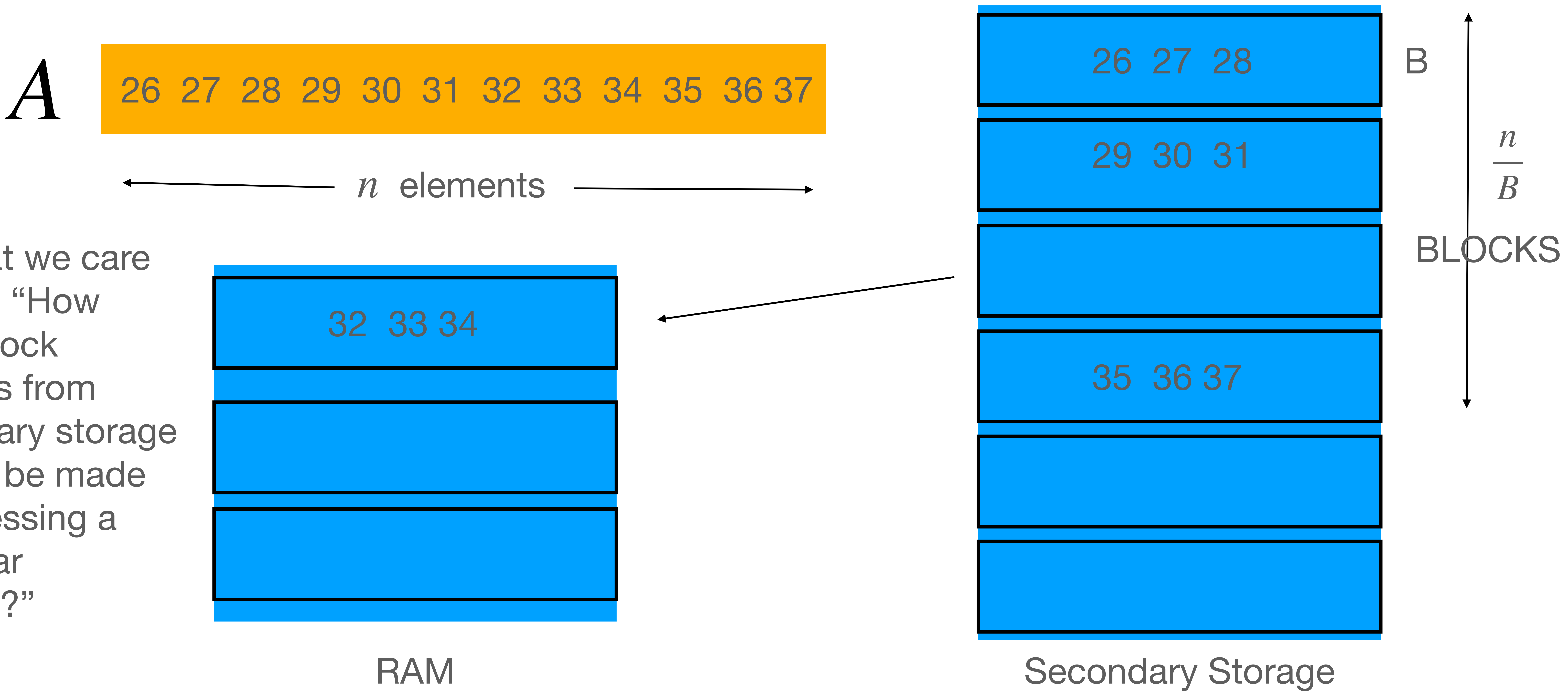


Efficient way to store data



Now, if we call search 34 now. Since 34 is in the RAM now, it takes so less time to search 34 in the RAM that we don't care about it.

Efficient way to store data



So, what we care about is "How many block transfers from Secondary storage need to be made for accessing a particular element?"

Efficient way to store data (Problem 2)

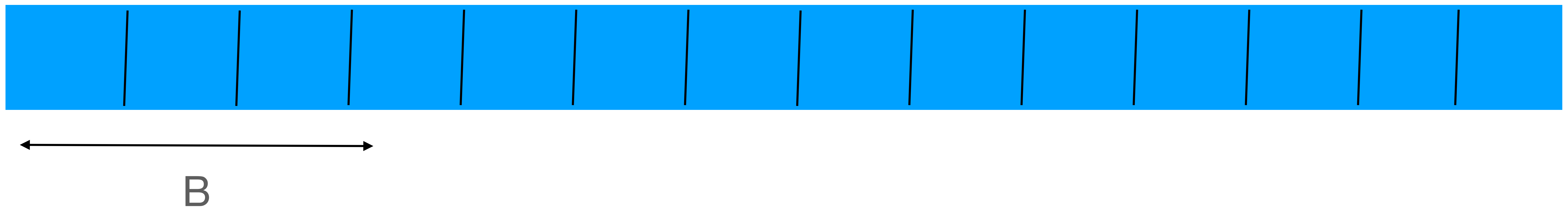
(a) Assume your data is stored on disk. Your data is a sorted array of size n, and spans many blocks. What is the number of block transfers needed (i.e. cost) of doing a linear search for an item? Leave your answer in terms of n and B.

(b) Assume your data is stored on disk. Your data is a sorted array of size n, and spans many blocks. What is the number of block transfers needed (i.e. cost) of doing a binary search for an item? Leave your answer in terms of n and B.

Efficient way to store data (Problem 2)

(a) Assume your data is stored on disk. Your data is a sorted array of size n, and spans many blocks. What is the number of block transfers needed (i.e. cost) of doing a linear search for an item? Leave your answer in terms of n and B.

(b) Assume your data is stored on disk. Your data is a sorted array of size n, and spans many blocks. What is the number of block transfers needed (i.e. cost) of doing a binary search for an item? Leave your answer in terms of n and B.



Since after transferring one block of data from secondary storage to RAM, we don't care about the searching time in that block so, we can consider the block of B elements as equivalent to 1 element in the array.

Efficient way to store data (Problem 2)

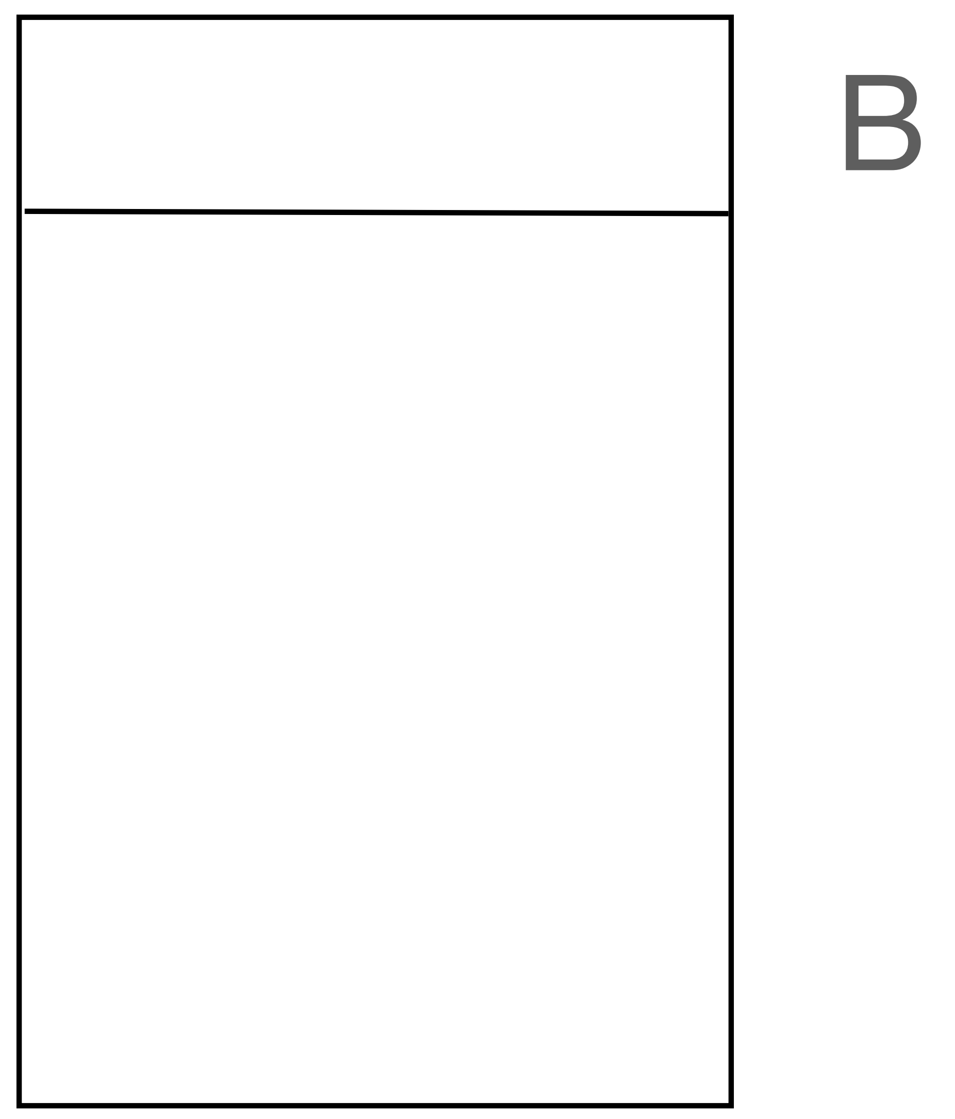
(a) Assume your data is stored on disk. Your data is a sorted array of size n, and spans many blocks. What is the number of block transfers needed (i.e. cost) of doing a linear search for an item? Leave your answer in terms of n and B.

$O(n/B)$. (Answer)

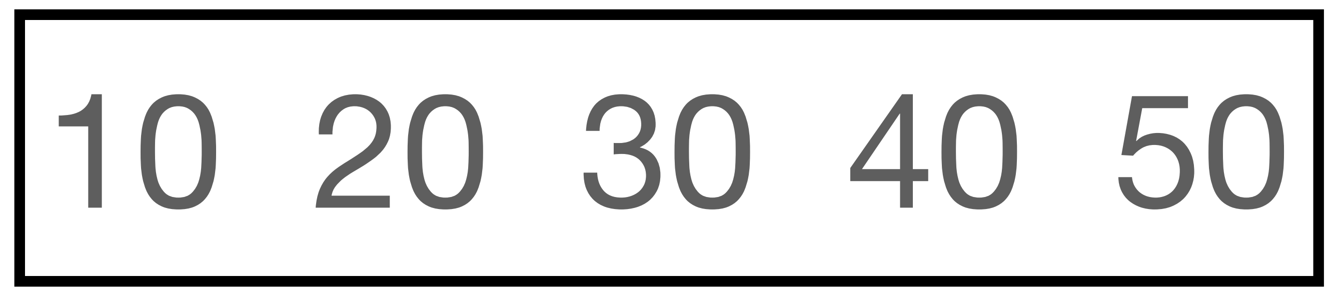
(b) Assume your data is stored on disk. Your data is a sorted array of size n, and spans many blocks. What is the number of block transfers needed (i.e. cost) of doing a binary search for an item? Leave your answer in terms of n and B.

$O(log(n/B))$. (Answer)

Efficient way to store data

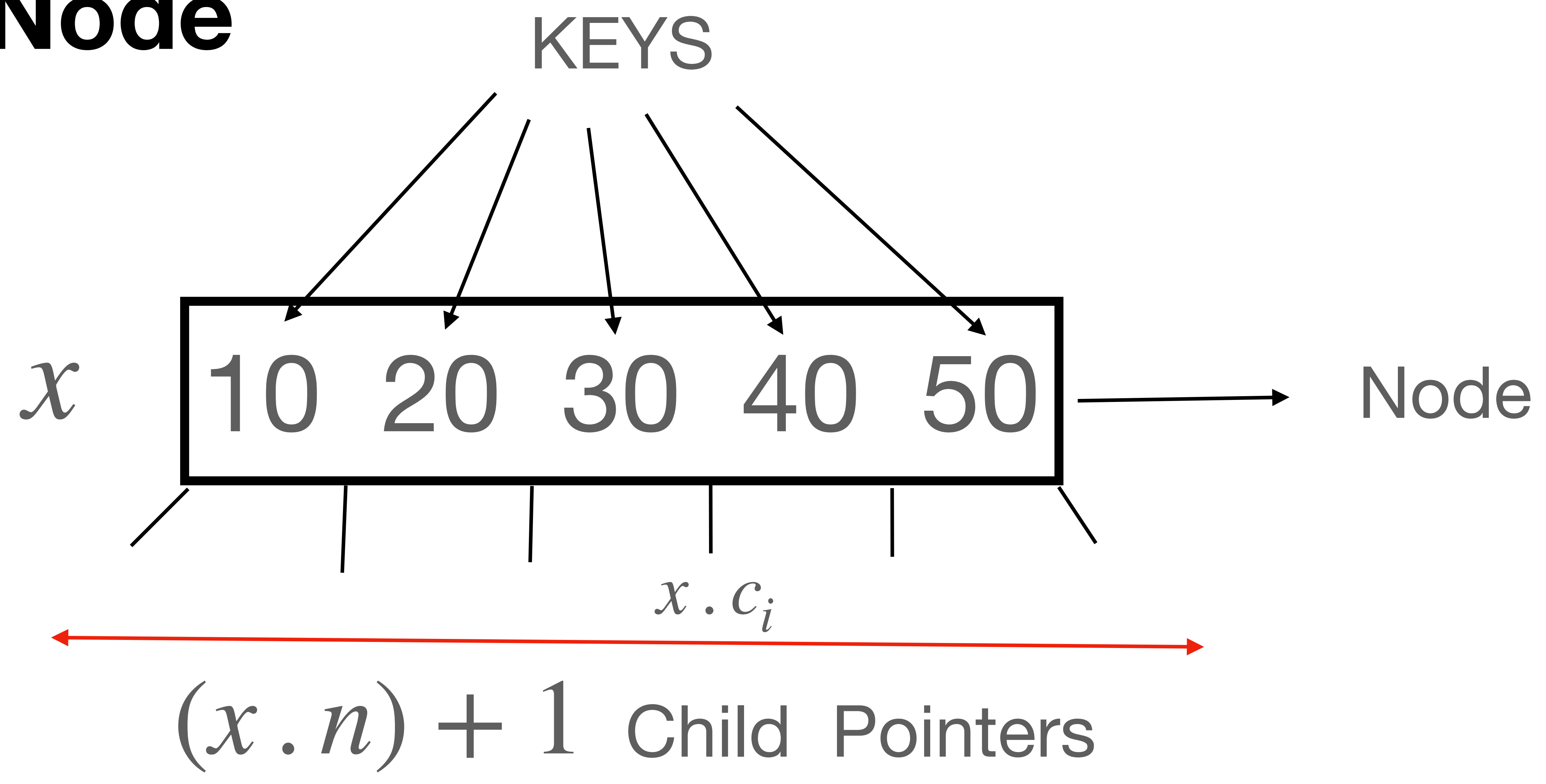


Memory



Create a node which has a lot of keys (data) in it.

(a, b) Trees
Attributes of a Node



$x.n$	Number of keys in the node x
$x.leaf$	Returns a boolean value TRUE if x is a leaf node, FALSE otherwise
$x.key_i$	Returns the value of the i^{th} key
$x.c_i$	Pointer to the i^{th} child node

(a, b) Trees
Rules

Rule 1: (a, b) child Policy

$(a - 1) \leq x.n \leq (b - 1)$ where $2 \leq a \leq (b + 1)/2$

Root node has a special treatment

$1 \leq root.n \leq (b - 1)$

All nodes except the leaf nodes has $((x.n) + 1))$ child nodes.

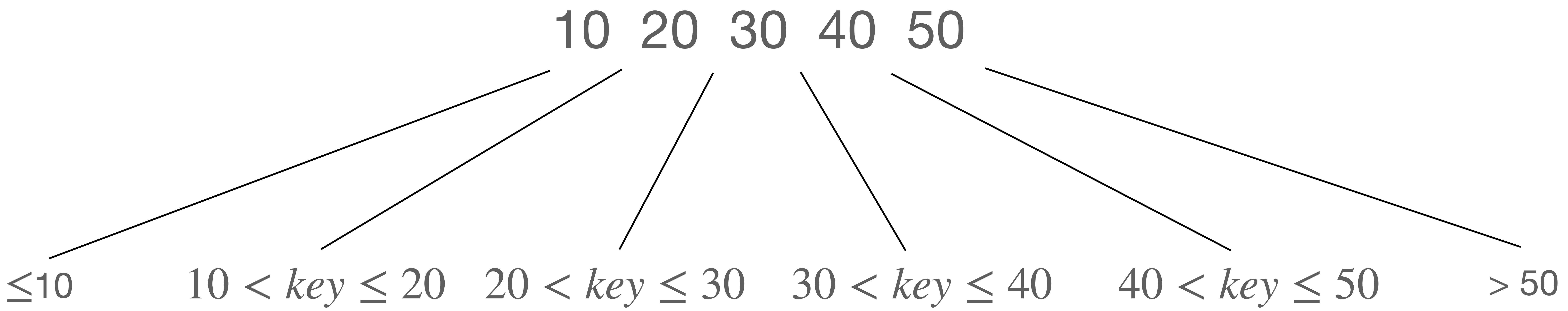
(a, b) **Trees**

Rules

Rule 2: Key ranges

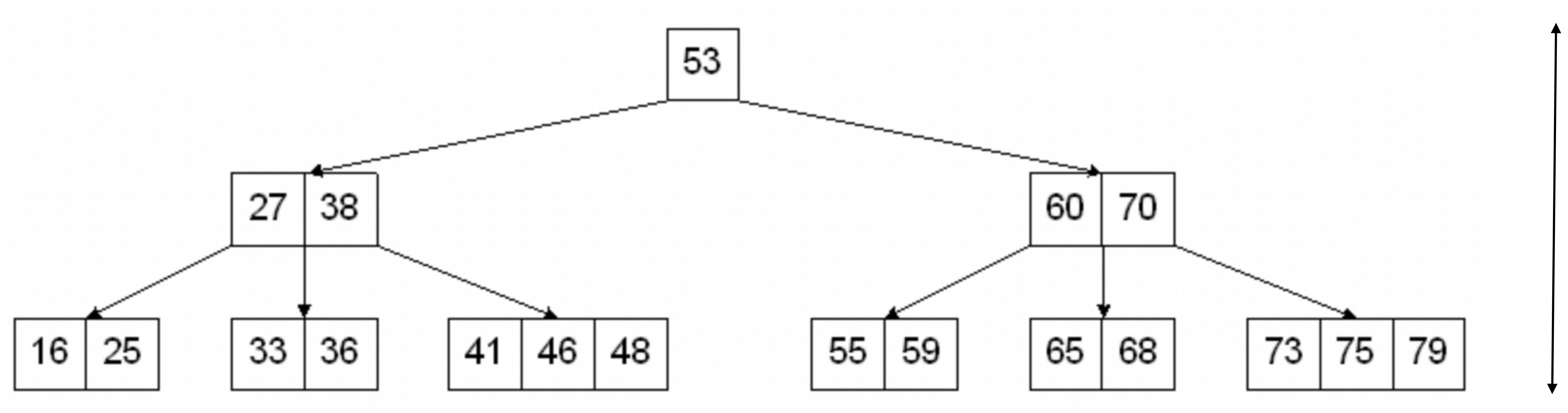
$$x\,.\,key_{i-1} < (x\,.\,c_i)\,.\,key \leq x\,.\,key_i$$

All keys are stored in sorted order in a node.



(a, b) Trees
Rules

Rule 3: Leaf nodes are at the same height



EQUAL HEIGHT

Balancing Rules (Problem 1(a))

Is (a, b) tree balanced ? If it is, which rules ensures that? If not why?

Balancing Rules (Problem 1(a))

Is (a, b) tree balanced ? If it is, which rules ensures that? If not why?

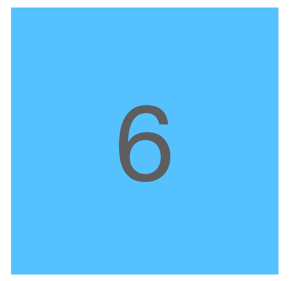
Both Rule 1 “(a, b) child Policy”,   and Rule 3 “ All leaves are at the same height ” makes it balanced.

Insertion

Insert the keys 6, 19, 17, 11 in a (2, 4) tree.

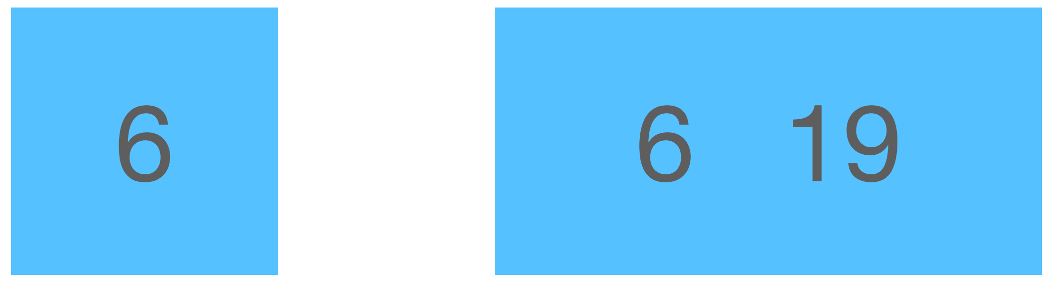
Insertion

Insert the keys 6, 19, 17, 11 in a (2, 4) tree.



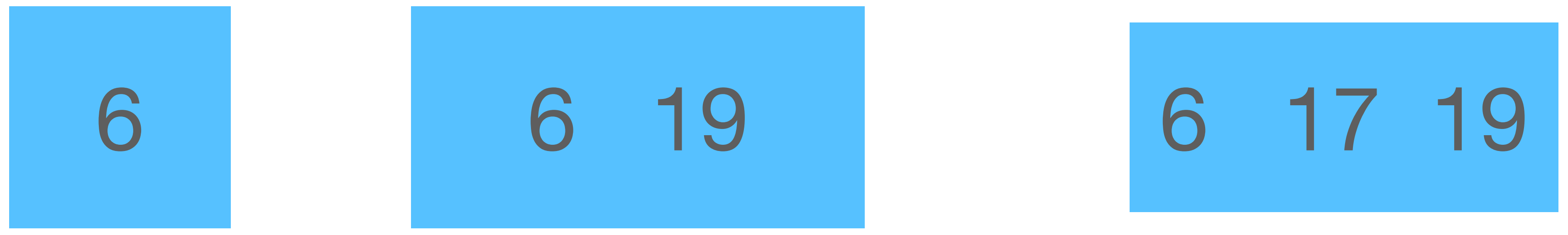
Insertion

Insert the keys 6, 19, 17, 11 in a (2, 4) tree.



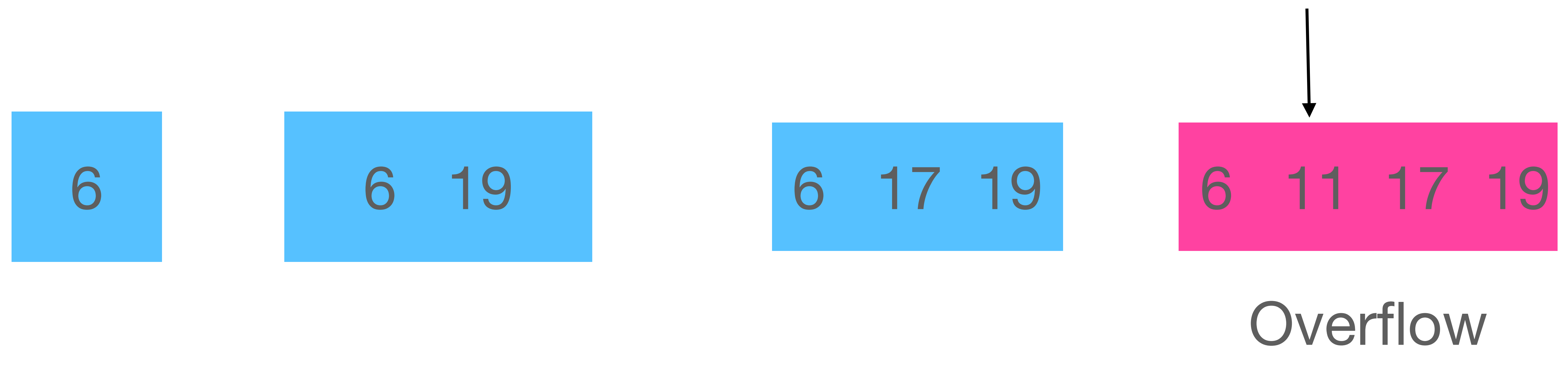
Insertion

Insert the keys 6, 19, 17, 11 in a (2, 4) tree.



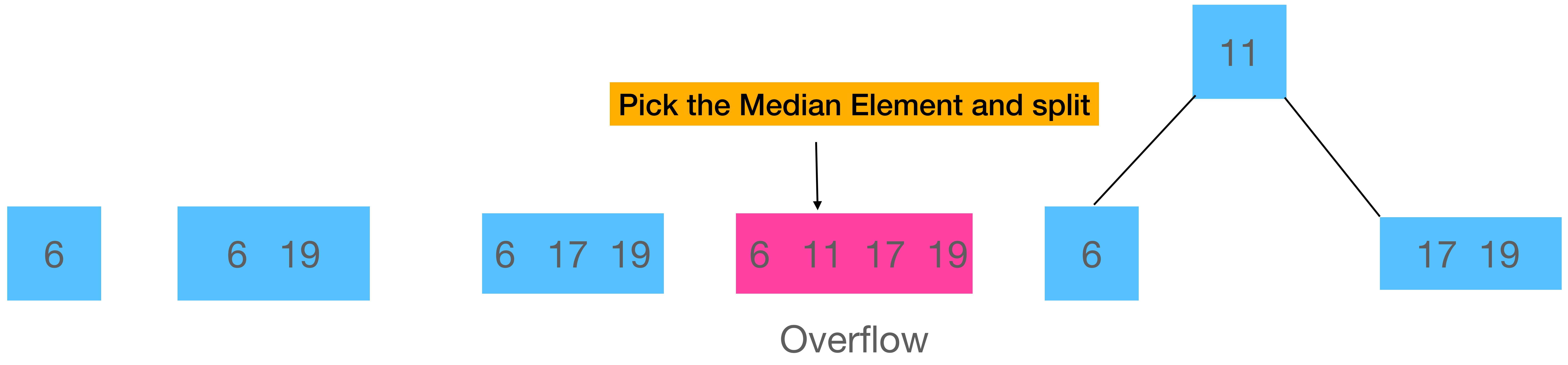
Insertion

Insert the keys 6, 19, 17, 11 in a (2, 4) tree.



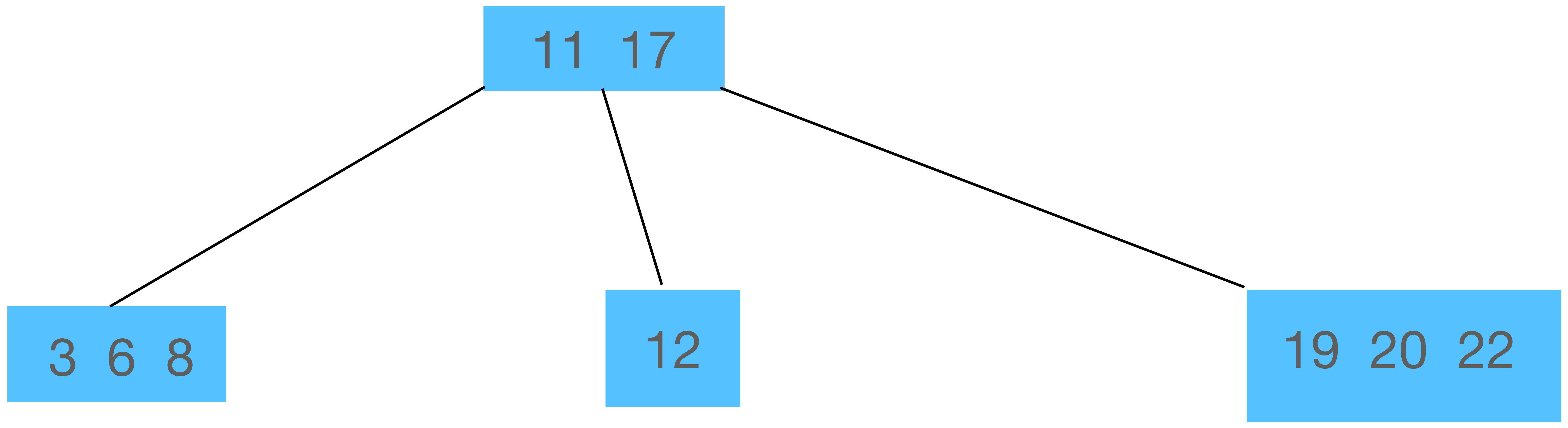
Insertion

Insert the keys 6, 19, 17, 11 in a (2, 4) tree.



Insertion

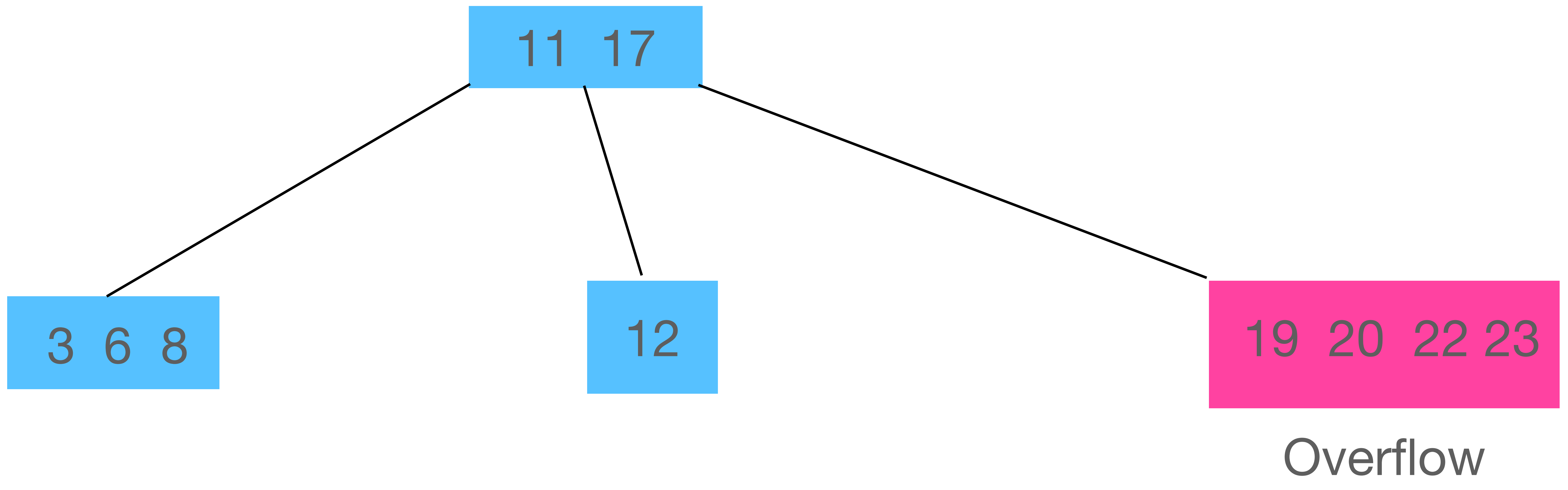
After inserting the keys 3, 12, 8, 20, 22 in the previous tree.



Now, insert 23

Insertion

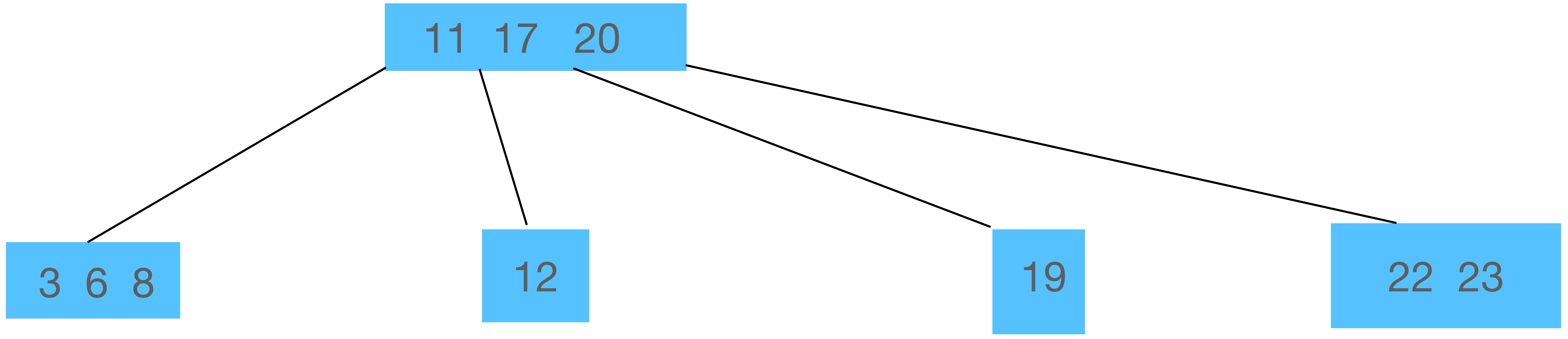
After inserting the keys 3, 12, 8, 20, 22 in the previous tree.



Now, insert 23

Insertion

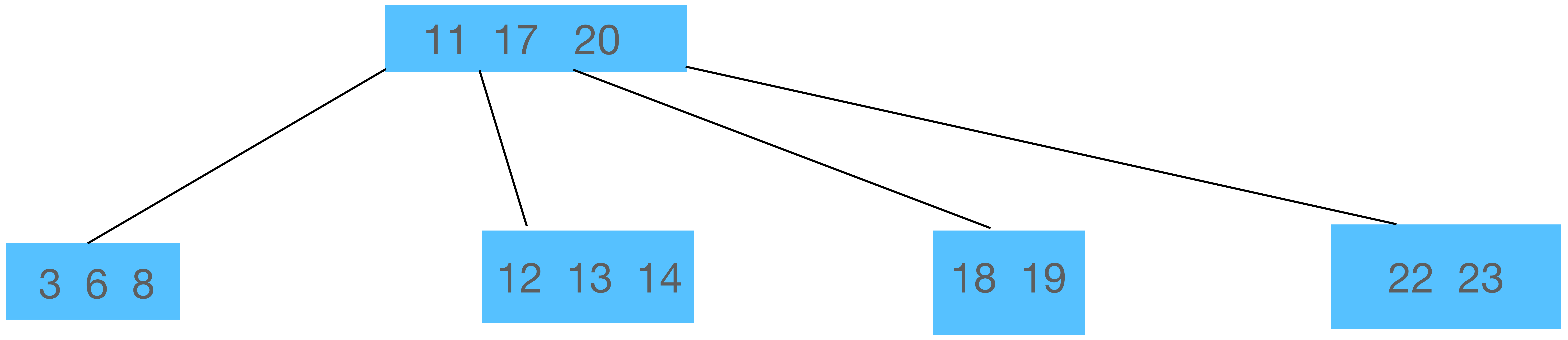
After inserting the keys 3, 12, 8, 20, 22 in the previous tree.



After inserting 23

Insertion

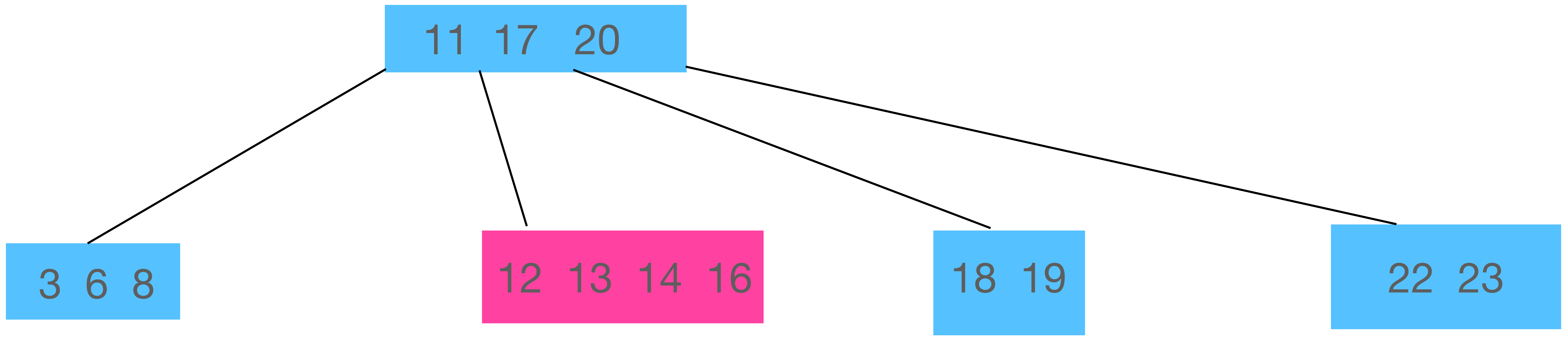
After inserting the keys 13, 14, 18 in the previous tree.



Now insert 16

Insertion

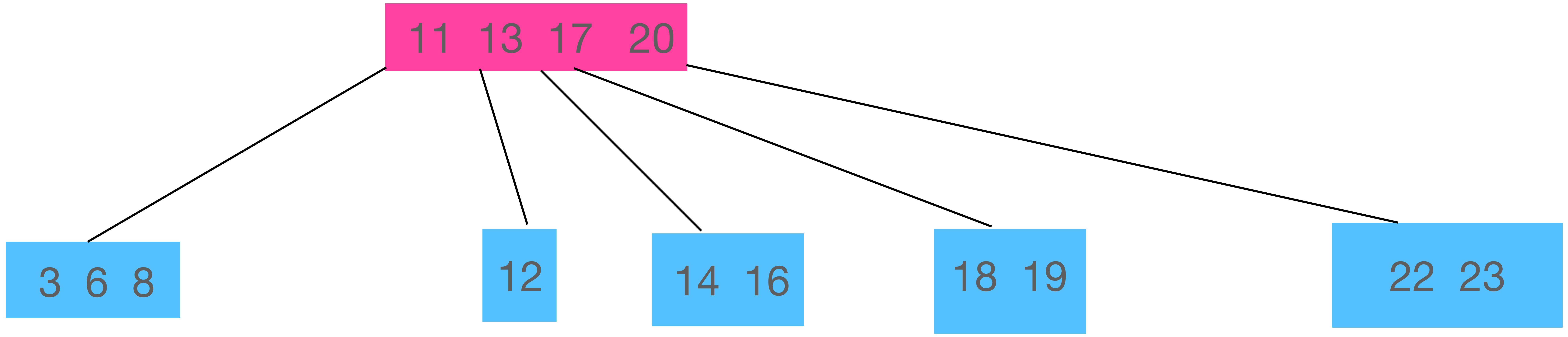
After inserting the keys 13, 14, 18 in the previous tree.



Now insert 16

Insertion

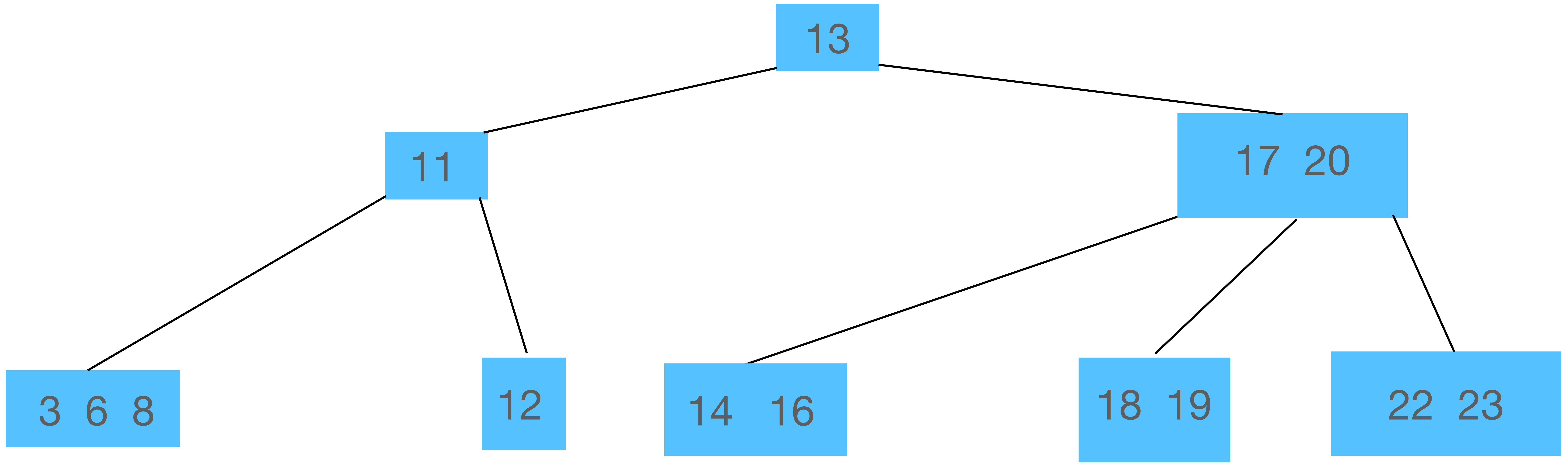
After inserting the keys 13, 14, 18 in the previous tree.



Now insert 16

Insertion

After inserting the keys 13, 14, 18 in the previous tree.



After inserting 16

Strategy for Insertion

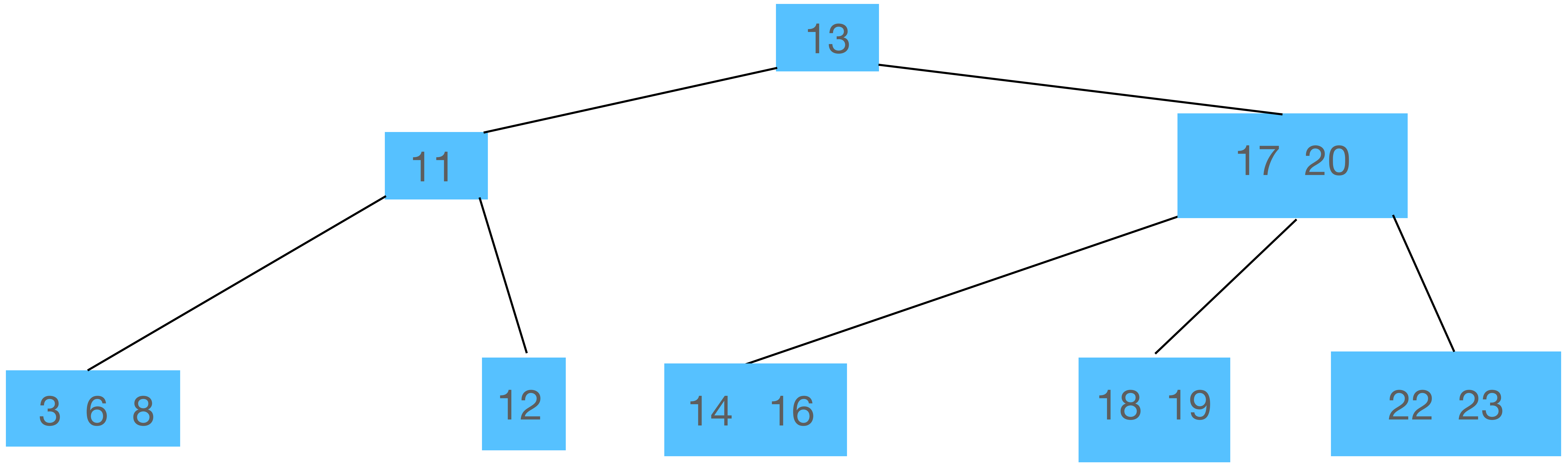
(1) At first put the element in its correct place in the node (which is in sorted order)

(2) If the node overflows, then find the median ($= \frac{b}{2}$ if b is even and $\frac{b+1}{2}$ if b is odd) key in that node.

(3) Put the median key in the parent and split the current node.

Searching

Search 18

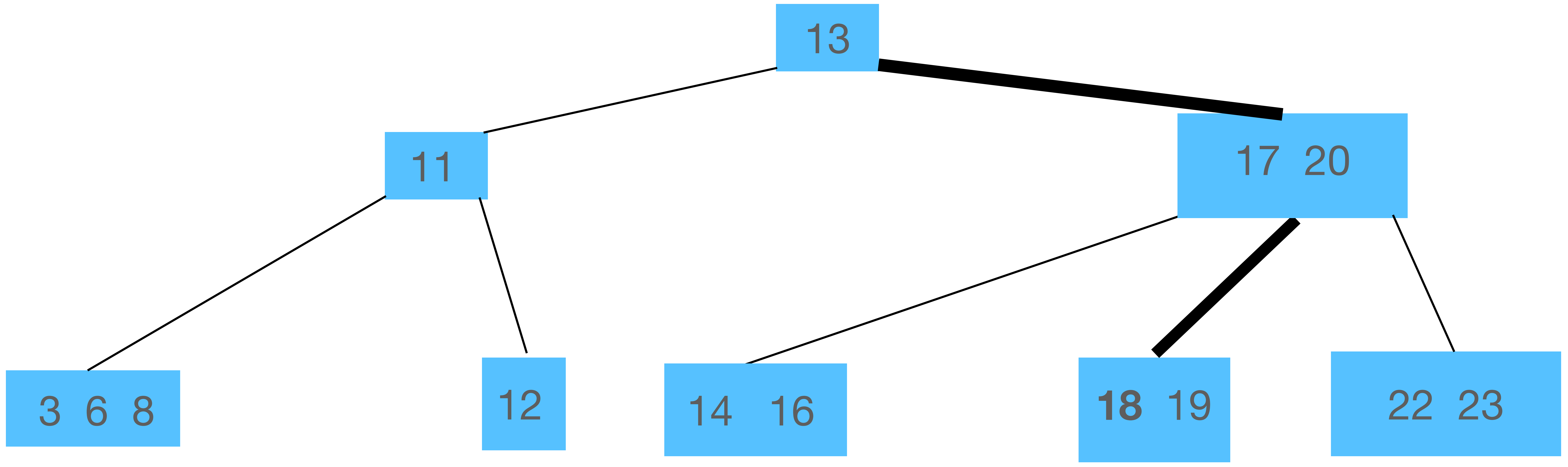


Searching

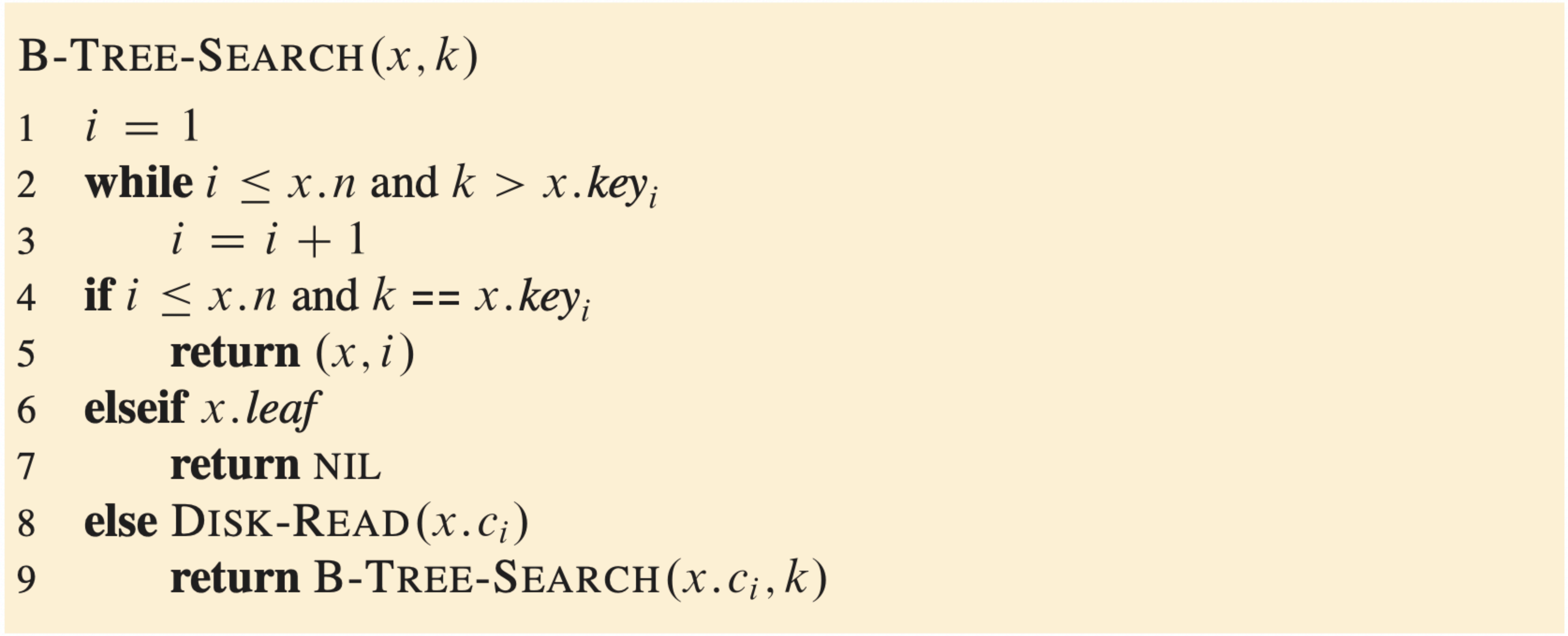
Search like we do in BST.

Only difference is we have to do either a linear search or binary search in each node to find the correct child pointer to traverse.

Search 18



Pseudocode of Searching (Problem 1(c))



```
B-TREE-SEARCH$(x, k)$
$i = 1$
while $i \le x.n$ and $k > x.key_i$
    $i = i + 1$
if $i \le x.n$ and $k == x.key_i$
    return $(x, i)$
elseif $x.leaf$
    return NIL
else DISK-READ$(x.c_i)$
    return B-TREE-SEARCH$(x.c_i, k)$
```

Deletion in (3,6) tree

Case 1: Deletion at the leaf

Delete 6

16
3 7 13
20 24
1 2
4 5 6
10 11 12
14 15
17 18 19
21 22
25 26

Deletion in (3,6) tree

Case 1: Deletion at the leaf

Delete 6

16
3 7 13
20 24
1 2
4 5
10 11 12
14 15
17 18 19
21 22
25 26

Deletion in (3,6) tree

Case 2: Problem with Orphan Children

Delete 13

16
3 7 13
20 24
1 2
4 5
10 11 12
14 15
17 18 19
21 22
25 26

Deletion in (3,6) tree

Case 2: Problem with Orphan Children

Delete 13

16
3 7
20 24
1 2
4 5
10 11 12
14 15
17 18 19
21 22
25 26

Deletion in (3,6) tree

Case 2: Problem with Orphan Children

Delete 13

16
3 7 12
20 24
1 2
4 5
10 11
14 15
17 18 19
21 22
25 26

Replace the parent key with the predecessor key which is the rightmost key in the left subtree or the successor key which is the leftmost key in the right subtree

Deletion in (3,6) tree

Case 3: Problem with Downflow

Delete 21

16
3 7
20 24
1 2
4 5
10 11 14 15
17 18 19
21 22
25 26

Deletion in (3,6) tree

Case 3: Problem with Downflow

Delete 21

16
3 7
20 24
1 2
4 5
10 11 14 15
17 18 19
22
25 26

Deletion in (3,6) tree

Case 3: Problem with Downflow

Delete 21

16
3 7
20 24
1 2
4 5
10 11 14 15
17 18 19
22
25 26

If at least one sibling has enough keys (at least ‘a’ keys) then it can donate to its key to the other sibling

Deletion in (3,6) tree

Case 3: Problem with Downflow

Delete 21

16
3 7
19 24
1 2
4 5
10 11 14 15
17 18
20 22
25 26

If at least one sibling has enough keys (at least 'a' keys) then it can donate to its key to the other sibling

Deletion in (3,6) tree

Case 3: Problem with Downflow

Delete 18

16
3 7
19 24
1 2
4 5
10 11 14 15
17 18
20 22
25 26

Deletion in (3,6) tree

Case 3: Problem with Downflow

Delete 18

16
3 7
19 24
1 2
4 5
10 11 14 15
17
20 22
25 26

Deletion in (3,6) tree

Case 3: Problem with Downflow

Delete 18

16
3 7
19 24
1 2
4 5
10 11 14 15
17
20 22
25 26

If the sibling does not have enough keys (at least 'a' keys) then merge the two siblings and the parent together

Deletion in (3,6) tree

Case 3: Problem with Downflow

Delete 18

16
3 7
24
1 2
4 5
10 11 14 15
17 19 20 22
25 26

If the sibling does not have enough keys (at least 'a' keys) then merge the two siblings and the parent together

Deletion in (3,6) tree

Case 3: Problem with Downflow

Delete 18

Downflow Again
16
3 7
24
1 2
4 5
10 11 14 15
17 19 20 22
25 26

If the sibling does not have enough keys (at least ‘a’ keys) then merge the two siblings and the parent together

Deletion in (3,6) tree

Case 3: Problem with Downflow

Delete 18

3 7 16 24
1 2
4 5
10 11 14 15
17 19 20 22
25 26

If the sibling does not have enough keys (at least ‘a’ keys) then merge the two siblings and the parent together

Strategy for Deletion

(1) At first delete the key at the node it is present currently.

(2) Now, detect does it have “the problem of orphan children” or “the problem of downflow”

(3) If it is “the problem of orphan children” then replace it with either the successor or predecessor key.

(4) if it is “the problem of down flow” then there are again two cases

(a) Its sibling (note that there can be two siblings left and right sibling) has enough (at least “a” keys) keys, then SHARE (look at the process of sharing) its key with the other sibling.

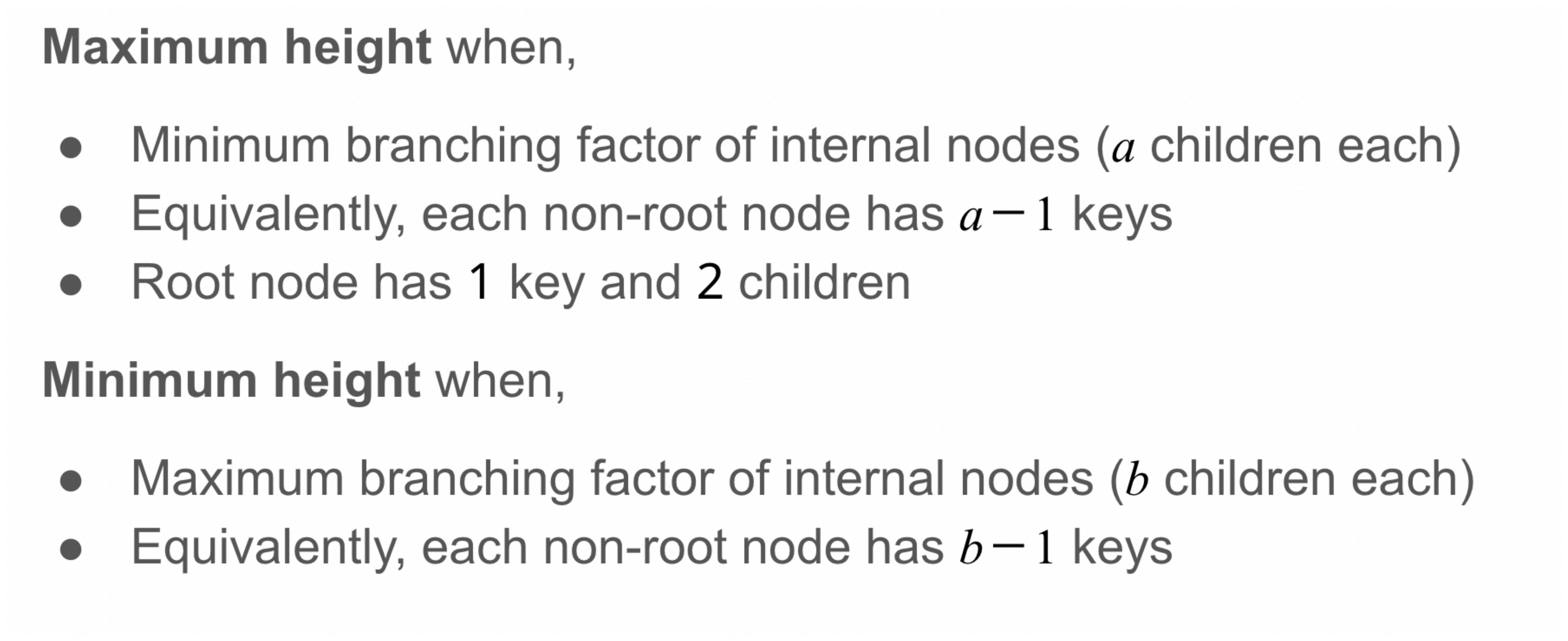
(b) Its sibling does not have enough keys then we MERGE (look how we merge).

Height of a B-tree (Problem 1(b))

What is the minimum and maximum height of a B-Tree with n keys?

Height of a B-tree

What is the minimum and maximum height of a B-Tree with n keys?



Maximum height when,

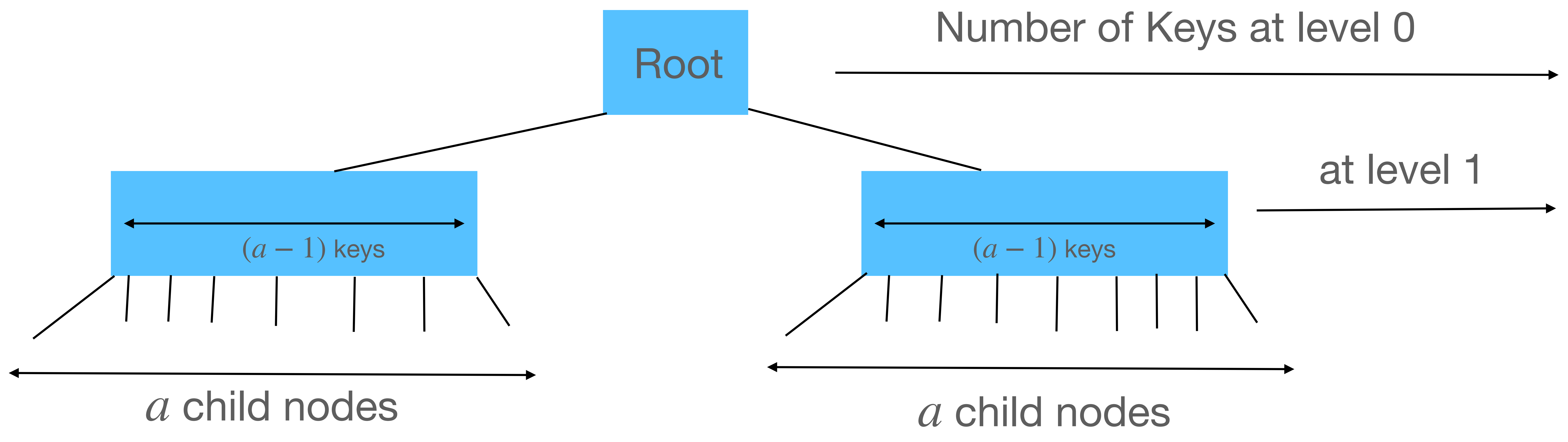
- Minimum branching factor of internal nodes (a children each)
- Equivalently, each non-root node has $a-1$ keys
- Root node has 1 key and 2 children

Minimum height when,

- Maximum branching factor of internal nodes (b children each)
- Equivalently, each non-root node has $b-1$ keys

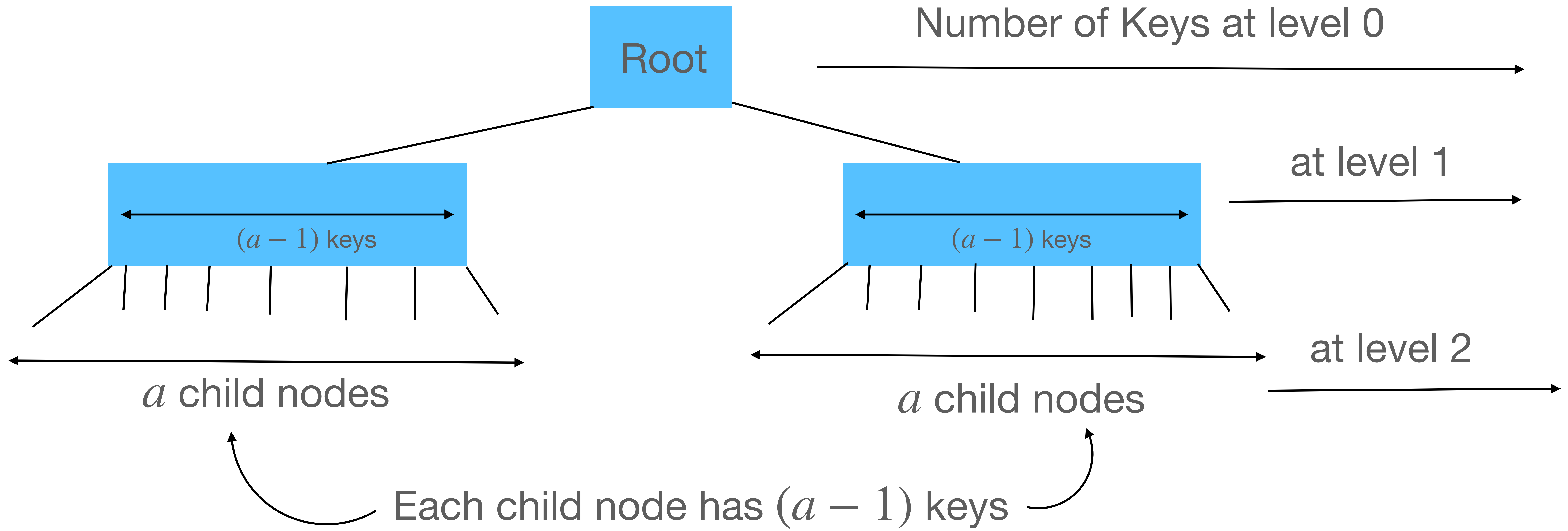
Height of a B-tree

Here, I am calculating the maximum height when the root node has two children



Number of Keys
1
$2(a-1)$

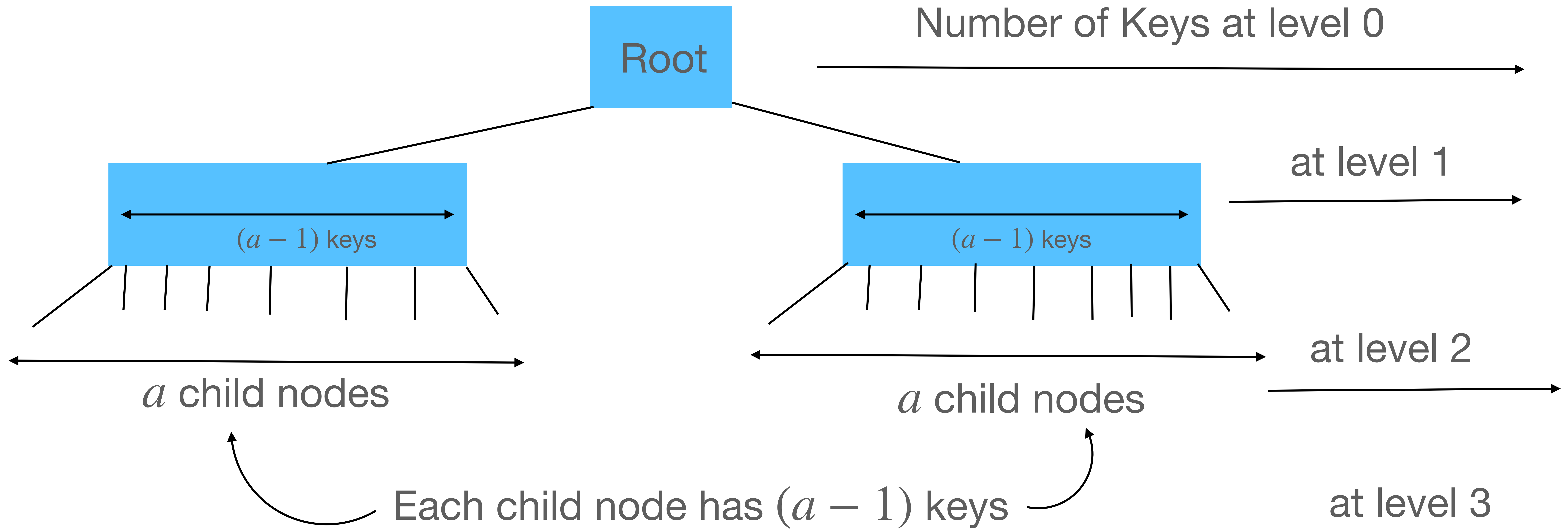
Height of a B-tree



Number of Keys
1
$2(a-1)$
$2a \times (a-1)$

Height of a B-tree

At height h we will have the number of keys = $2a^{h-1} \times (a-1)$



Number of Keys
1
$2(a-1)$
$2a \times (a-1)$
$2a^2 \times (a-1)$

Height of a B-Tree

So, the total number of keys = $1 + (a - 1) \times (2 + 2a + 2a^2 + 2a^3 + \ldots + 2a^{h-1})$ = n

Height of a B-Tree

So, the total number of keys = $1 + (a-1) \times (2 + 2a + 2a^2 + 2a^3 + \ldots + 2a^{h-1})$ = n

$$\implies 1 + 2(a-1) \times (1 + a + a^2 + a^3 + \ldots + a^{h-1}) = n$$

$$\implies 1 + 2(a-1) \times \frac{a^h - 1}{a - 1} = n$$

$$\implies 1 + 2 \times (a^h - 1) = n$$

$$\implies 2a^h - 1 = n$$

$$h = log_a(n)$$

Using similar calculations we can calculate the minimum height which is $log_b(n)$

Cost of searching the keylist, splitting, merging, sharing

Problem 2(c)

What is the cost of searching a keylist in a B-tree node?

What is the cost of splitting a B-tree node?

What is the cost of merging or sharing B-tree nodes?

Cost of searching the keylist, splitting, merging, sharing

Problem 2(c)

What is the cost of searching a keylist in a B-tree node?

What is the cost of splitting a B-tree node?

What is the cost of merging or sharing B-tree nodes?

Answer: They are all $\mathrm{O}(1)$ since they require some constant number of block transfers followed by operation on the block in-memory.

Cost of Searching, Insertion, Deletion

Problem 2(d)

What is the cost of searching a B-tree? What is the cost of inserting or deleting in a B-tree?

These are all $O(log_a n)$ because

1. Each node is contained in a block. i.e., the cost only depends on the height of the tree.
2. Cost of the in-memory operations on a single node is only O(1).

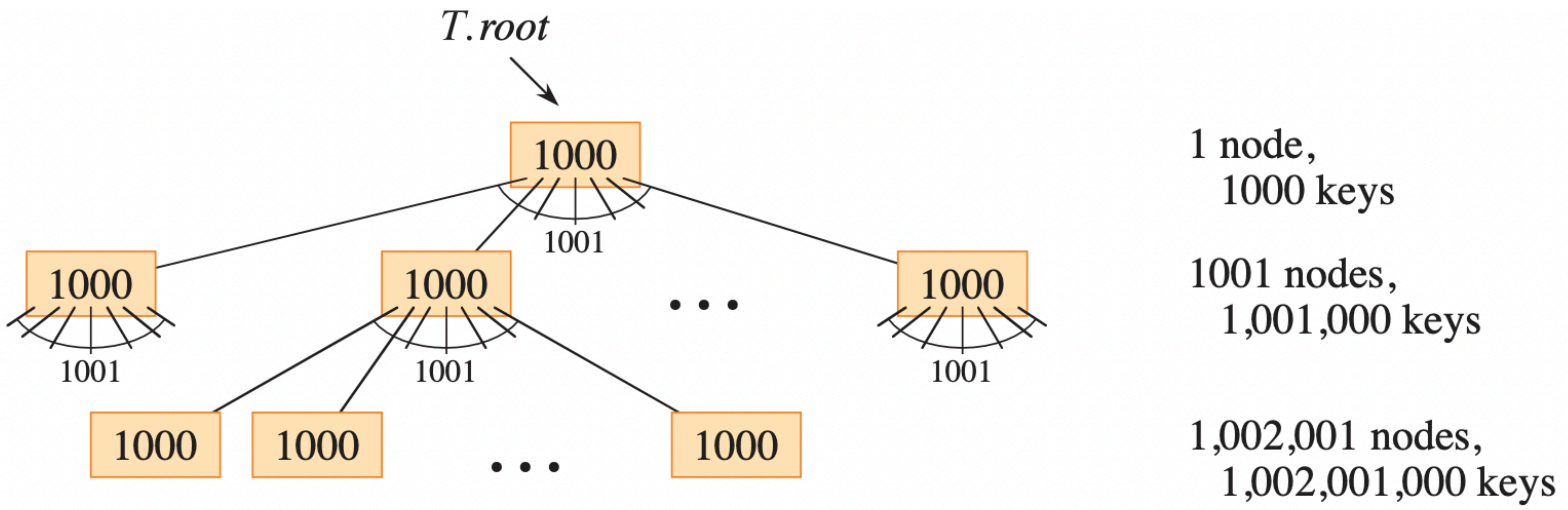
Cost of Searching, Insertion, Deletion

Problem 1(d), 1(g), 1(h)

What is the cost of searching a B-tree? What is the cost of inserting or deleting in a B-tree?

Normally, we do not care about the time taken to search an element when the block is in the RAM but suppose now we care about that time also, then the time taken would be $O(log_a n) \times O(log_2 b)$ because we are doing a binary search within the block to search an element, insert in the correct position and delete the correct element.

1 Billion Data in only 3 levels!!!



Thank You