B+-trees

Perhaps the most widely used index ever!

CMSC 420

B+-trees were not covered in Spring 2019, yet it is very important that you go through these slides so that you can understand the real story behind <K,V> pairs, the nature of a range query, and why it's an excellent idea to lower the height of your tree-based index so that you can index into more disk-resident data with a smaller spatial cost in memory!

The true story: <K, V> pairs

- So far, we have only cared about storing **`Comparable`**s such that:
  - **Search** is optimized.
  - **Insert** is optimized.
  - **Delete** is optimized.
- We have required that all those elements are **`Comparable`**s because… we need to compare them.

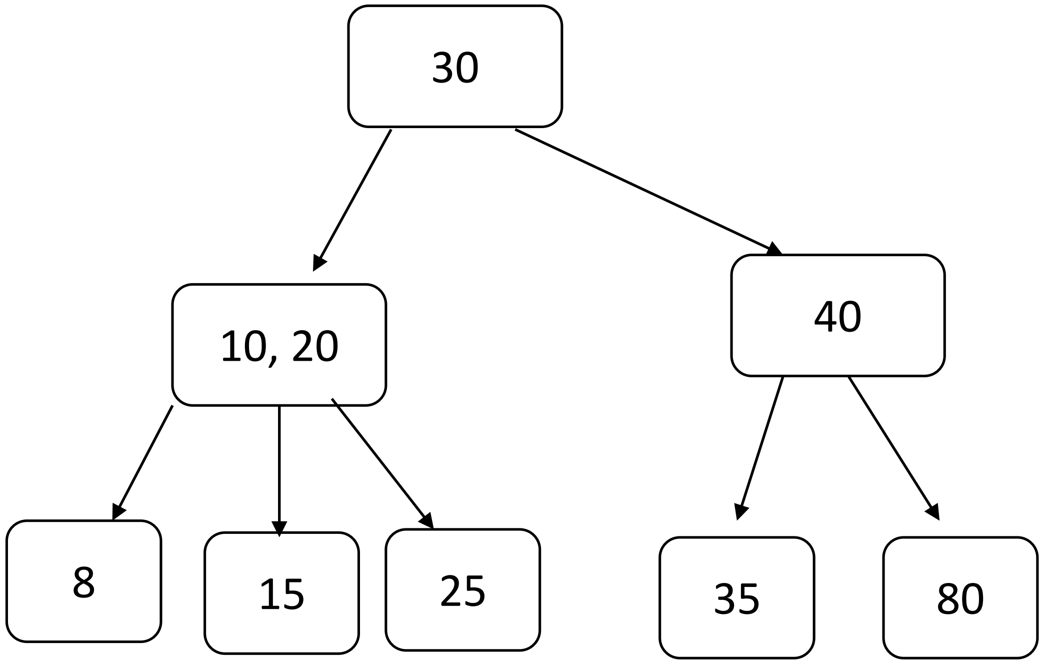
The true story: <K, V> pairs

- So far, we have only cared about storing **`Comparable`**s such that:
  - **Search** is optimized.
  - **Insert** is optimized.
  - **Delete** is optimized.
- We have required that all those elements are **`Comparable`**s because… we need to compare them.
- **But does this mean that you can only store things that have a 1-1 mapping with the naturals?**

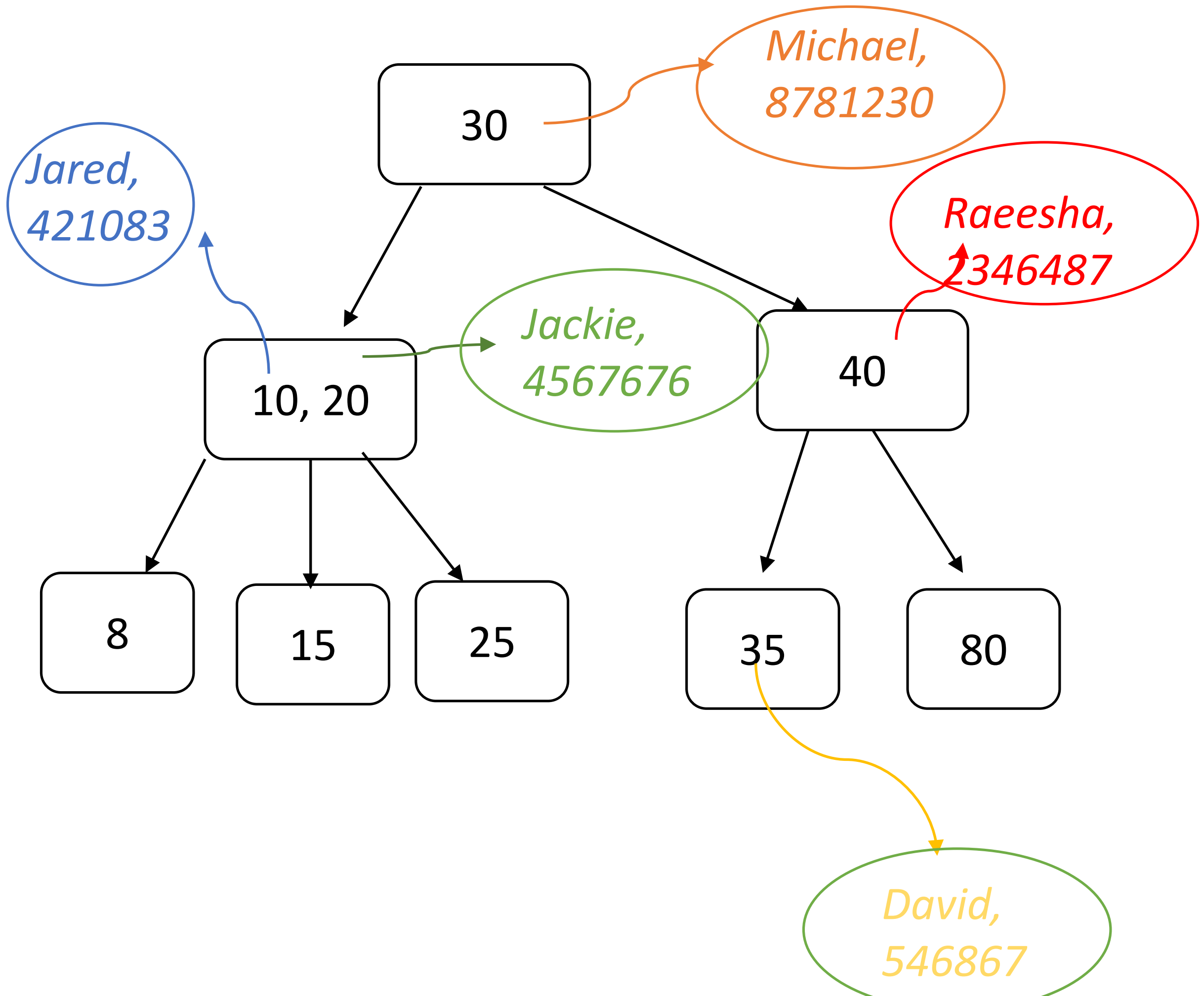
The true story: <K, V> pairs

- **Clearly not**, we can store images, files, database table records…
- **The true story is as follows:** We have been dealing only with Comparable **keys,** where what we might want is a **complex data value.**
- By employing a sufficiently large key dictionary (e.g integers), we can associate every value V with some unique key K. **It is assumed that K points to V in $\mathcal{O}(1)$!** *(Fair assumption* ***as long as we are in memory****)*
- **All** our work on data structures so far **has concerned this organization of keys**, since once we find a key, we can access the associated value in $\mathcal{O}(1)$ ☺

The true story: <K, V> pairs

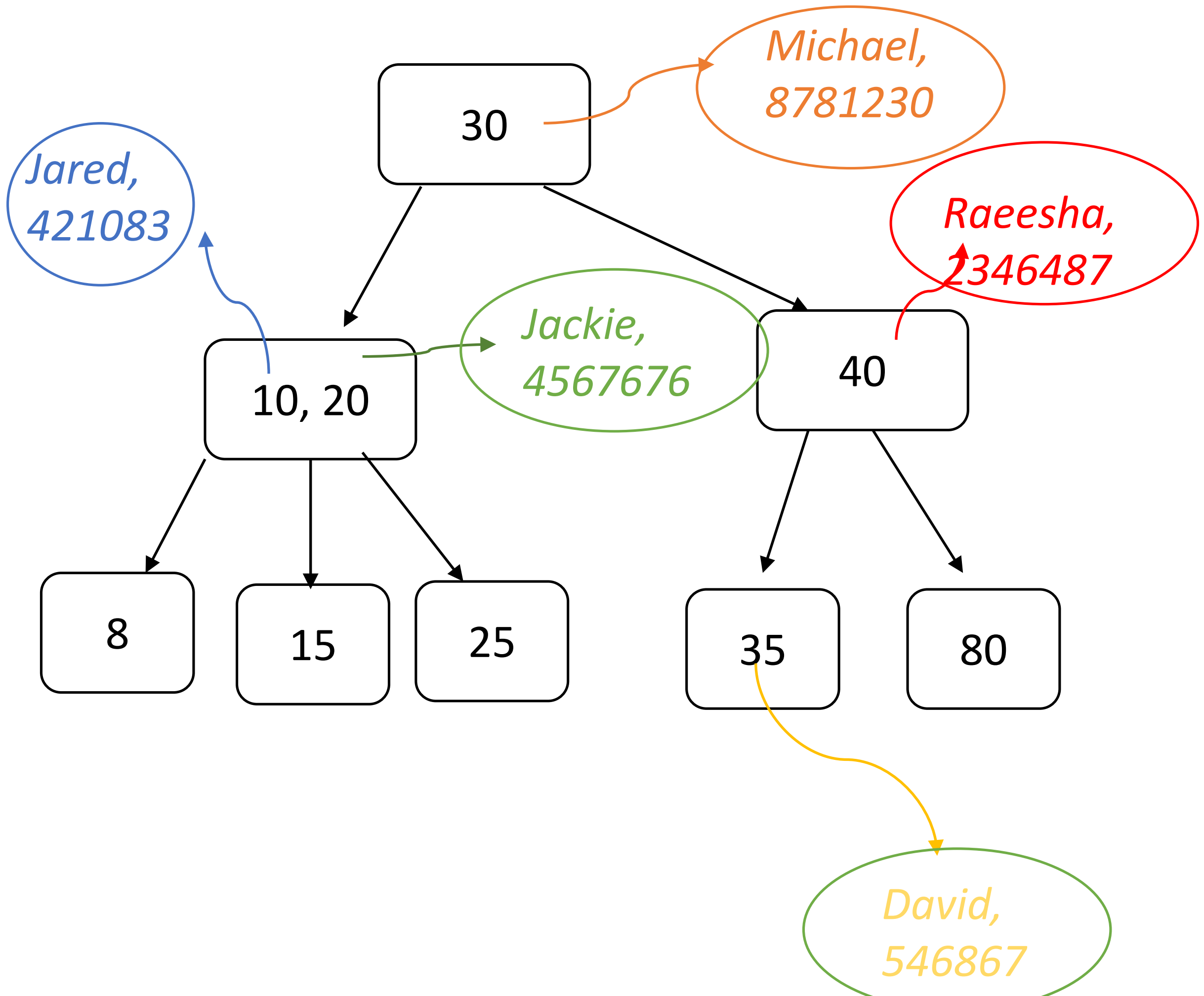


The true story: <K, V> pairs



ID	Name	UID
10	Jared	421083
…	…	…
20	Jackie	4567676
…	…	…
30	Michael	8781230
…	…	…
35	David	546867
…	…	…
40	Raeesha	2346487
…	…	…

The true story: <K, V> pairs



ID	Name	UID
10	Jared	421083
…	…	…
20	Jackie	4567676
…	…	…
30	Michael	8781230
…	…	…
35	David	546867
…	…	…
40	Raeesha	2346487
…	…	…

$\underbrace{\text{Name} \quad \text{UID}}_{\text{This is what we want (the value)}}$

Relational Databases

ID	Name	UID	Major	Campus resident	Rooms with	Parking space ID	Year
10	Jared	421083	POLI	N	NULL	89343	3
…	…	…					
20	Jackie	4567676	NULL	Y	40	243654	1
…	…	…					
30	Michael	8781230	ENGR	N	NULL	23465	4
…	…	…					
35	David	546867	NULL	Y	NULL	NULL	1
…	…	…	…	…	…	…	…
40	Raeesha	2346487	CS	Y	20	NULL	2
…	…	…					

Relational Databases

ID	Name	UID	Major	Campus resident	Rooms with	Parking space ID	Year
10	Jared	421083	POLI	N	NULL	89343	3
…	…	…					
20	Jackie	4567676	NULL	Y	40	243654	1
…	…	…					
30	Michael	8781230	ENGR	N	NULL	23465	4
…	…	…					
35	David	546867	NULL	Y	NULL	NULL	1
…	…	…	…	…	…	…	…
40	Raeesha	2346487	CS	Y	20	NULL	2
…	…	…					

This entire table is on disk!

Relational Databases

ID	Name	UID	Major	Campus resident	Rooms with	Parking space ID	Year
10	Jared	421083	POLI	N	NULL	89343	3
…	…	…					
20	Jackie	4567676	NULL	Y	40	243654	1
…	…	…					
30	Michael	8781230	ENGR	N	NULL	23465	4
…	…	…					
35	David	546867	NULL	Y	NULL	NULL	1
…	…	…	…	…	…	…	…
40	Raeesha	2346487	CS	Y	20	NULL	2
…	…	…					

This entire table is on disk!

Speaking of disks….

Rotational Hard disks and pages

- Disk space is divided into so-called **pages.**
  - Typical size: **4KB**
- **PULLING PAGES FROM DISK TAKES A LOT OF TIME!**
  - Fastest seek time for enterprise HDDs: **4ms**
  - **Orders of magnitude worse** than RAM (in the μs) and registers/cache (in the ns)
- **Page Fault:** Access of data pointed to by our program, which of course runs in main memory, but the data itself **resides on disk.**
- Don't confuse with cache miss: an address sought by our program was not in cache, but might be found in main memory.

Rotational Hard disks and pages

- Operating Systems allow applications to allocate buffers for reading and writing *EXACTLY* **PAGE_SIZE** kilobytes big.
  - *Usually, PAGE_SIZE=4KB*
- The application can then do whatever it needs with the data. If a change of the data needs to persist on disk, the **entire page** will be flushed to disk.
  - Even if exactly one byte was changed.
- Beneficial for buffers to **persist in memory** if possible (leverage locality), to avoid going back to disk.

Rotational Hard disks and pages

- Disk fragmentation: Under or over-utilization of disk pages by an application.
  - Under-utilization: pages are largely empty and space is wasted.
  - Over-utilization: the application takes up a lot of new hard disk pages, and indices have to be updated (this costs time)
- We will not talk about how we can control and optimize disk space (beyond scope).

Example Queries

TABLE STUDENTS

ID	***Name***	***UID***	***Major***	***Campus resident***	***Rooms with***	***Parking space ID***	***Year***
10	*Jared*	*421083*	*POLI*	*N*	*NULL*	*89343*	*3*
…	…	…					
20	*Jackie*	*4567676*	*NULL*	*Y*	*40*	*243654*	*1*
…	…	…					
30	*Michael*	*8781230*	*ENGR*	*N*	*NULL*	*23465*	*4*
…	…	…					
35	*David*	*546867*	*NULL*	*Y*	*NULL*	*NULL*	*1*
…	…	…	…	…	…	…	…
40	*Raeesha*	*2346487*	*CS*	*Y*	*20*	*NULL*	*2*
…	…	…					

1. Show me all fields from **all** records in the table . (`SELECT * from STUDENTS`)

Example Queries

TABLE STUDENTS

ID	Name	UID	Major	Campus resident	Rooms with	Parking space ID	Year
10	Jared	421083	POLI	N	NULL	893439	3
…	…	…					
20	Jackie	4567676	NULL	Y	40	243654	1
…	…	…					
30	Michael	8781230	ENGR	N	NULL	234656	4
…	…	…					
35	David	546867	NULL	Y	NULL	NULL	1
…	…	…	…	…	…	…	…
40	Raeesha	2346487	CS	Y	20	NULL	2
…	…	…					

4KB

4KB

4KB

1. Show me all fields from **all** records in the table . (`SELECT * from STUDENTS`)

- Brute-force: Pay seek cost sc for all the page faults ($\lceil {}^{N}/_{4096} \rceil$ for N bytes of disk space required for table).
- Also, pay exactly $r = {}^{N}/_{rec\_size}$ total time for printing every single record.
- Only good news: Note that you don't have to allocate N bytes of main memory for the result of this query: when you've printed the data of an entire page, use the same buffer for the next page (no need to write anything to disk either).

Example Queries

`TABLE STUDENTS`

ID	Name	UID	Major	Campus resident	Rooms with	Parking space ID	Year
10	Jared	421083	POLI	N	NULL	893439	3
…	…	…					
20	Jackie	4567676	NULL	Y	40	243654	1
…	…	…					
30	Michael	8781230	ENGR	N	NULL	234656	4
…	…	…					
35	David	546867	NULL	Y	NULL	NULL	1
…	…	…	…	…	…	…	…
40	Raeesha	2346487	CS	Y	20	NULL	2
…	…	…					

CAN WE DO BETTER?

Yes

No

1. Show me all fields from **all** records in the table . (`SELECT * from STUDENTS`)

- Brute-force: Pay seek cost sc for all the page faults ($\lceil N/_{4096} \rceil$ for N bytes of disk space required for table).
- Also, pay exactly $r = {}^{N}/_{rec\_size}$ total time for printing every single record.
- Only good news: Note that you don't have to allocate N bytes of main memory for the result of this query: when you've printed the data of an entire page, use the same buffer for the next page (no need to write anything to disk either).

Example Queries

TABLE STUDENTS

ID	Name	UID	Major	Campus resident	Rooms with	Parking space ID	Year
10	Jared	421083	POLI	N	NULL	893439	3
…	…	…					
20	Jackie	4567676	NULL	Y	40	243654	1
…	…	…					
30	Michael	8781230	ENGR	N	NULL	234656	4
…	…	…					
35	David	546867	NULL	Y	NULL	NULL	1
…	…	…	…	…	…	…	…
40	Raeesha	2346487	CS	Y	20	NULL	2
…	…	…					

CAN WE DO BETTER?

Yes No

- This query asks for **everything** and **anything**!
- Sooner or later, we **will** pay the price!
- Even if you build a Red-Black Tree over IDs… you still **have to scan all pages**!

1. Show me all fields from **all** records in the table . (`SELECT * from STUDENTS`)

- Brute-force: Pay seek cost sc for all the page faults ($\lceil {}^{N}/_{4096} \rceil$ for N bytes of disk space required for table).
- Also, pay exactly $r = {}^{N}/_{rec\_size}$ total time for printing every single record.
- Only good news: Note that you don't have to allocate N bytes of main memory for the result of this query: when you've printed the data of an entire page, use the same buffer for the next page (no need to write anything to disk either).

Example Queries

TABLE STUDENTS

ID	Name	UID	Major	Campus resident	Rooms with	Parking space ID	Year
10	Jared	421083	POLI	N	NULL	893439	3
…	…	…					
20	Jackie	4567676	NULL	Y	40	243654	1
…	…	…					
30	Michael	8781230	ENGR	N	NULL	234656	4
…	…	…					
35	David	546867	NULL	Y	NULL	NULL	1
…	…	…	…	…	…	…	…
40	Raeesha	2346487	CS	Y	20	NULL	2
…	…	…					

2. Show me all fields from **students ID is between 740 and 1043**. (`SELECT * from STUDENTS WHERE ID >=740 AND ID <= 1043`)

Example Queries

TABLE STUDENTS

ID	Name	UID	Major	Campus resident	Rooms with	Parking space ID	Year
10	Jared	421083	POLI	N	NULL	893439	3
…	…	…					
20	Jackie	4567676	NULL	Y	40	243654	1
…	…	…					
30	Michael	8781230	ENGR	N	NULL	234656	4
…	…	…					
35	David	546867	NULL	Y	NULL	NULL	1
…	…	…	…	…	…	…	…
40	Raeesha	2346487	CS	Y	20	NULL	2
…	…	…					

CAN WE DO BETTER NOW?

Yes

Foreshadowing does, like, nothing for me

2. Show me all fields from **students ID is between 740 and 1043**. (`SELECT * from STUDENTS WHERE ID >=740 AND ID <= 1043`)

Example Queries

TABLE STUDENTS

We will need an index over IDs!

ID	Name	UID	Major	Campus resident	Rooms with	Parking space ID	Year
10	Jared	421083	POLI	N	NULL	893439	3
…	…	…					
20	Jackie	4567676	NULL	Y	40	243654	1
…	…	…					
30	Michael	8781230	ENGR	N	NULL	234656	4
…	…	…					
35	David	546867	NULL	Y	NULL	NULL	1
…	…	…	…	…	…	…	…
40	Raeesha	2346487	CS	Y	20	NULL	2
…	…	…					

Yes

Foreshadowing does, like, nothing for me

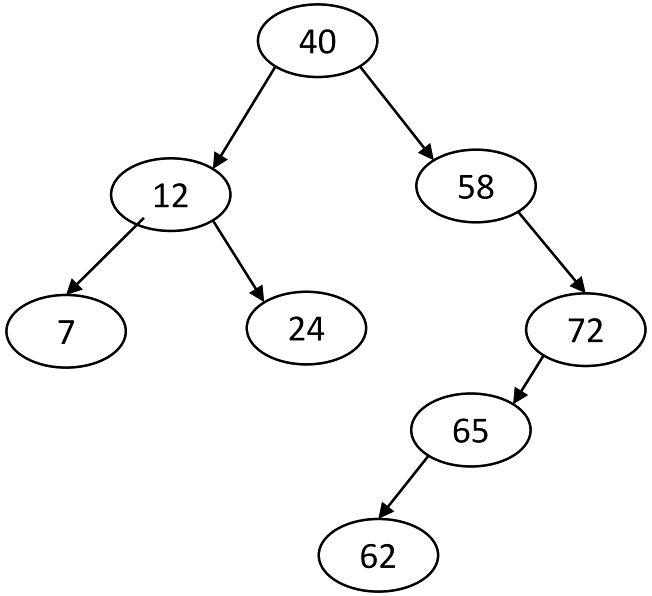
2. Show me all fields from **students ID is between 740 and 1043**. (`SELECT * from STUDENTS WHERE ID >=740 AND ID <= 1043`)

Warmup Exercise

- Take 10 to implement the **void** method

`rangeSearch(Node n, int min, int max, List<Integer> list)`,

such that we can perform range search on this BST!



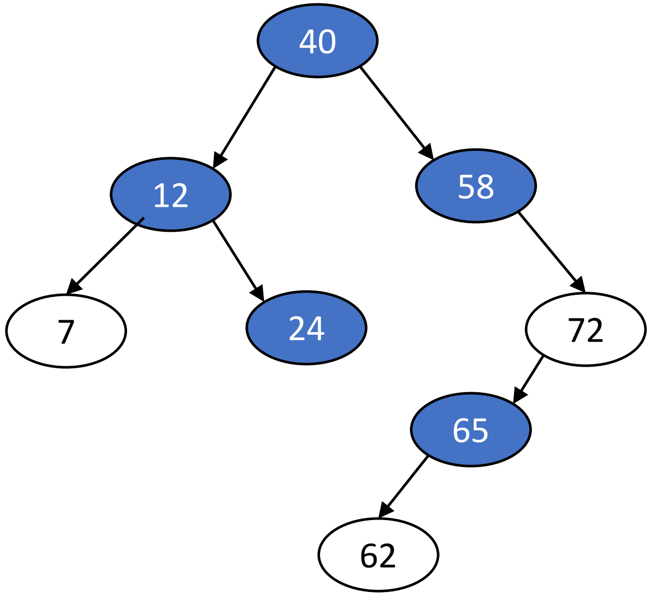
Warmup Exercise

- Take 10 to implement the **void** method

  `rangeSearch(Node n, int min, int max, List<Integer> list)`,

  such that we can perform range search on this BST!

[12, 65]



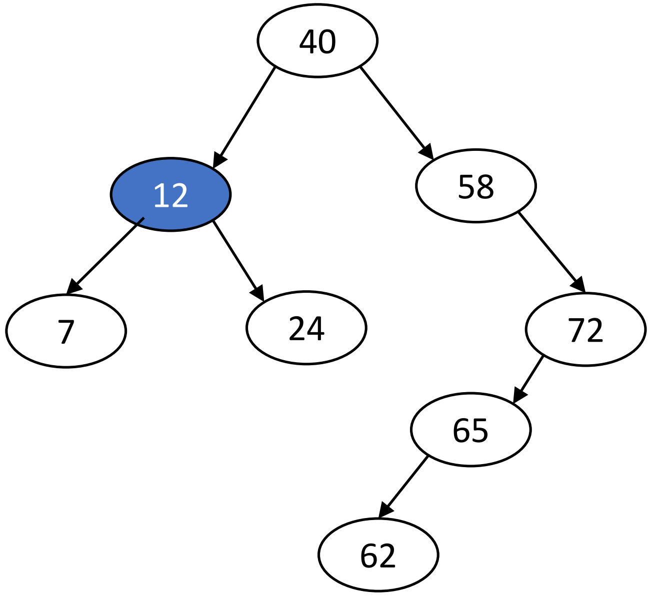
Warmup Exercise

- Take 10 to implement the **void** method

  `rangeSearch(Node n, int min, int max, List<Integer> list)`,

  such that we can perform range search on this BST!

[12, 12]



Warmup Exercise

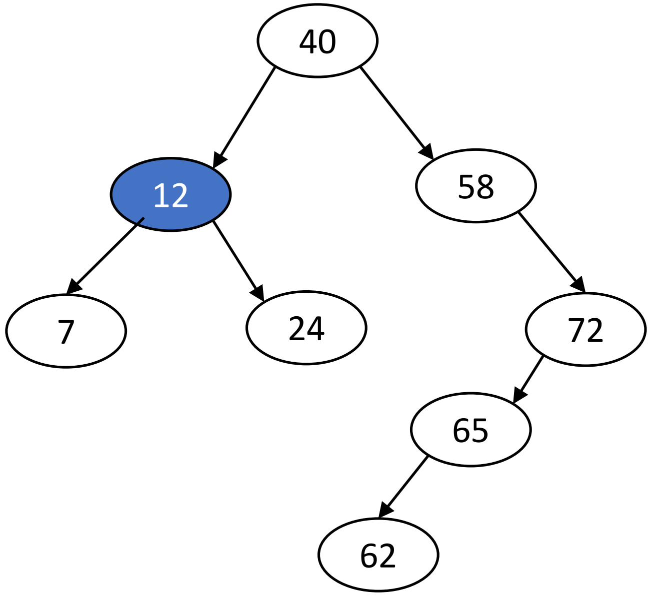
- Take 10 to implement the **void** method

`rangeSearch(Node n, int min, int max, List<Integer> list)`,

such that we can perform range search on this BST!

[11, 12]

(11 is not a key in this tree…)



Warmup Exercise

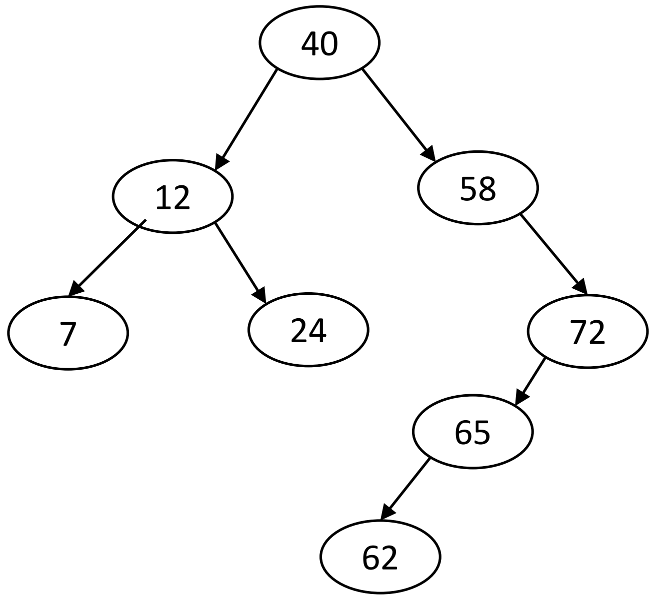
- Take 10 to implement the **void** method

  `rangeSearch(Node n, int min, int max, List<Integer> list)`,

  such that we can perform range search on this BST!

[70, 20]

Just throw your favorite exception!



Warmup Exercise

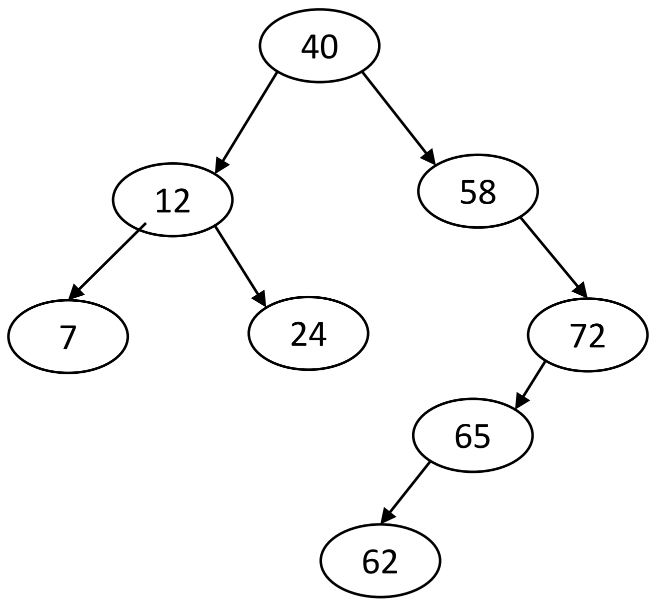
- Take 10 to implement the **void** method

`rangeSearch(Node n, int min, int max, List<Integer> list)`,

such that we can perform range search on this BST!

Example Call:

```
ArrayList<T> list = new ArrayList();
rangeSearch(root, min, max, list)
```



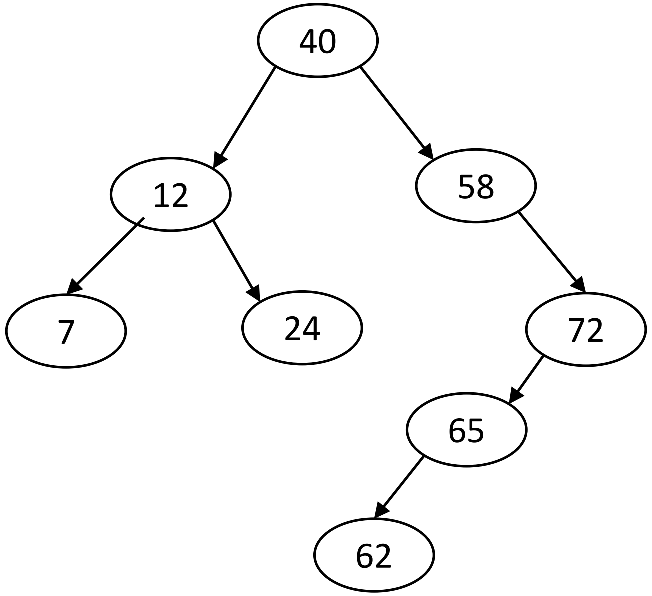
Warmup Exercise

- Take 10 to implement the **void** method

rangeSearch(Node n, int min, int max, List<Integer> list),

such that we can perform range search on this BST!

```
// This will fail if min and max are null, but whatever
private void rangeSearch(Node n, T min, T max, List<T> list){
    assert min.compareTo(max) <= 0 : "Min ought to be smaller than or equal to max.";
    if(n == null)
        return;
    if(n.key.compareTo(min) > 0 && n.key.compareTo(max) < 0){
        rangeSearch(n.left, min, n.key, list);
        list.add(n.key);
        rangeSearch(n.right, n.key, max, list);
    } else if(n.key.compareTo(min) == 0) {
        list.add(n.key);
        rangeSearch(n.right, n.key, max, list);
    } else if(n.key.compareTo(min) < 0) {
        rangeSearch(n.right, n.key, max, list);
    } else if(n.key.compareTo(max) == 0 ) {
        rangeSearch(n.left, min, n.key, list);
        list.add(n.key); // ! If you want the range sorted, the addition should be
after the recursive call.
    } else if(n.key.compareTo(max) > 0 ) {
        rangeSearch(n.left, min, n.key, list);
    }
}
```



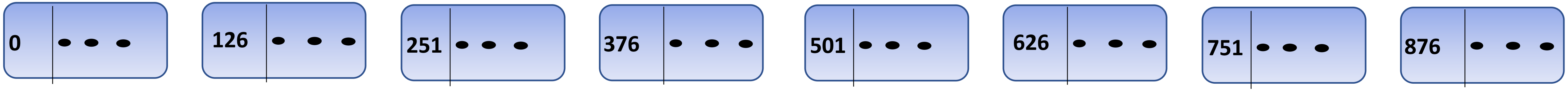
Key-Value Dictionaries are database indices

- We can do this with a BST. We know how to do range search, so…
- Let’s assume that we build our index as a BST…

Key-Value Dictionaries are database indices

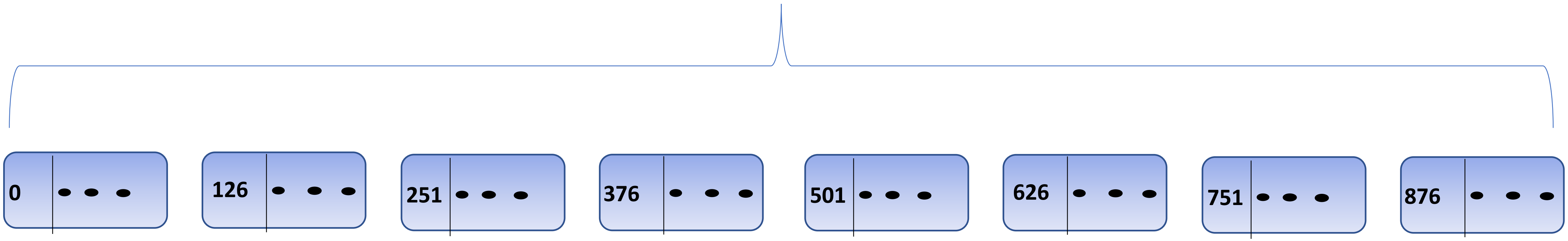
- We can do this with a BST. We know how to do range search, so…
- Let's assume that we build our index as a BST…
  - A bottom-up approach seems natural!
- Assume 1000 records of 32 bytes each.
  - $\frac{1000 \times 32\ B}{4KB} = 8$ pages
- Key: BUILDING the dictionary is expensive.
  - But once it's in place, **reading** and **writing** data, even when it lies on disk, will be much faster than having nothing in place.

Building a BST index, bottom-up



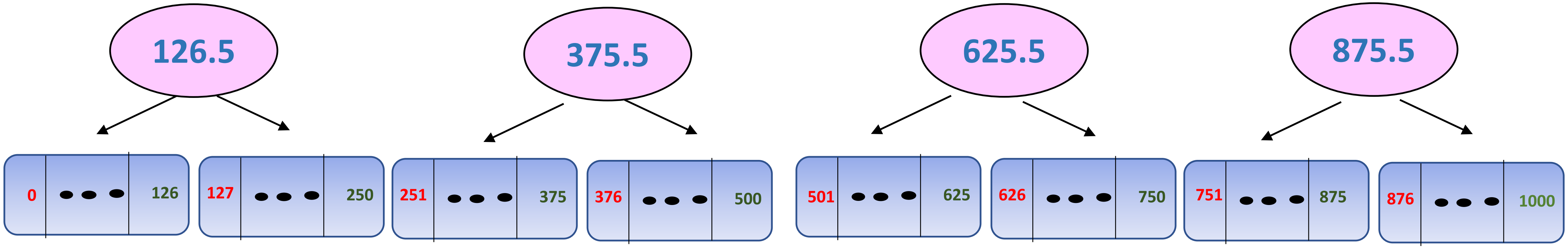
Building a BST index, bottom-up

All of those are on disk!



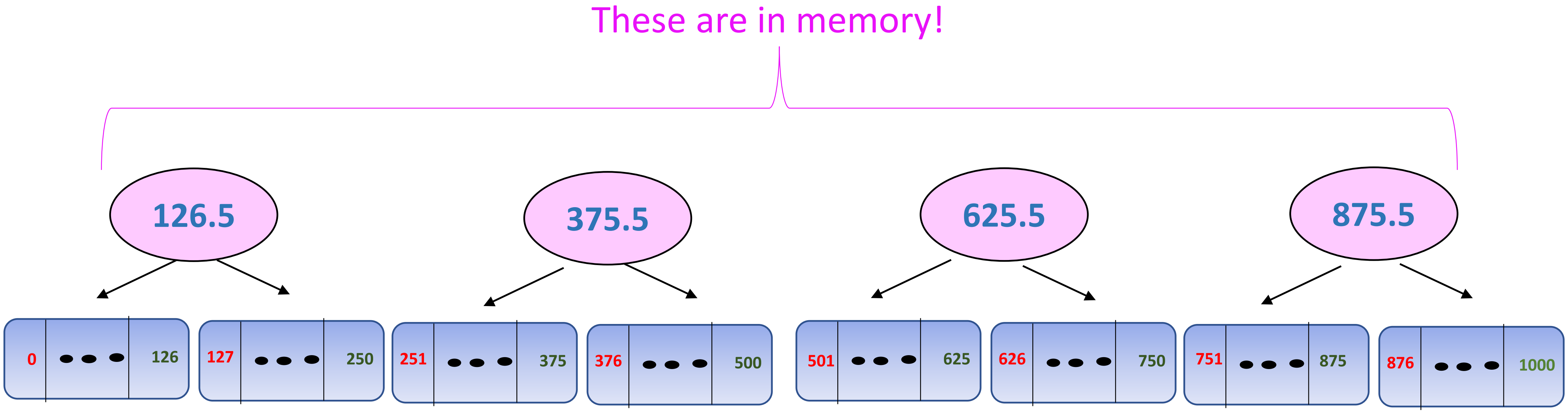
Building a BST index, bottom-up

- To separate the disk pages, we need an appropriate separator.
- Answer: Take the **average** of the **first** and **last** ids of every page!
  - ➢ Other values could also work, e.g median. Average is just easy.



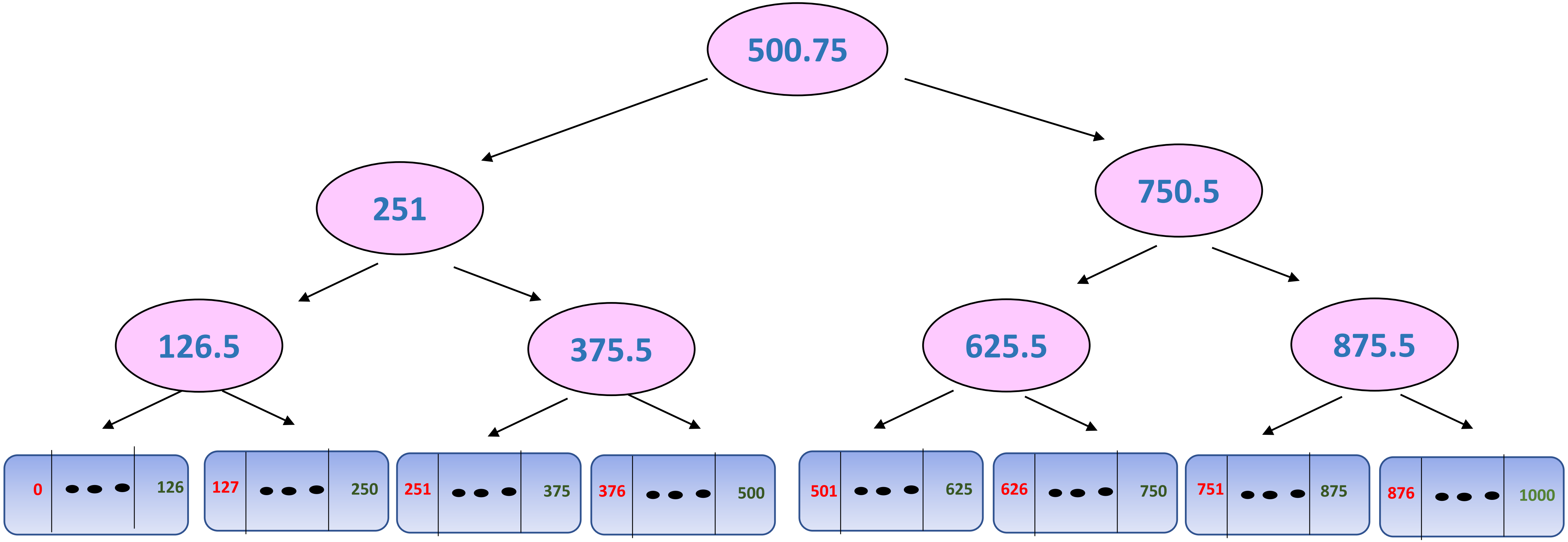
Building a BST index, bottom-up

- To separate the disk pages, we need an appropriate separator.
- Answer: Take the **average** of the **first** and **last** ids of every page!
  - ➢ Other values could also work, e.g median. Average is just easy.



Building a BST index, bottom-up

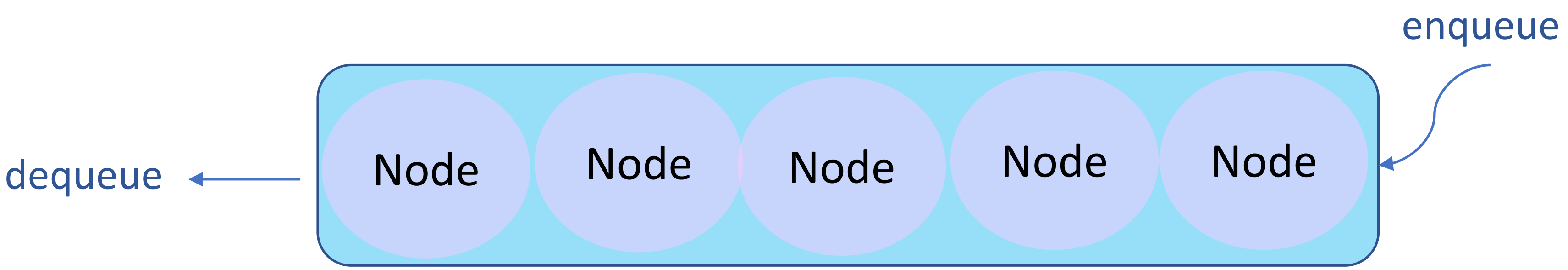
- We can repeat the exact same process for the rest of the nodes!



Some code that does this

```
class Node {
    T key;
    Node left, right;
}
```

Node

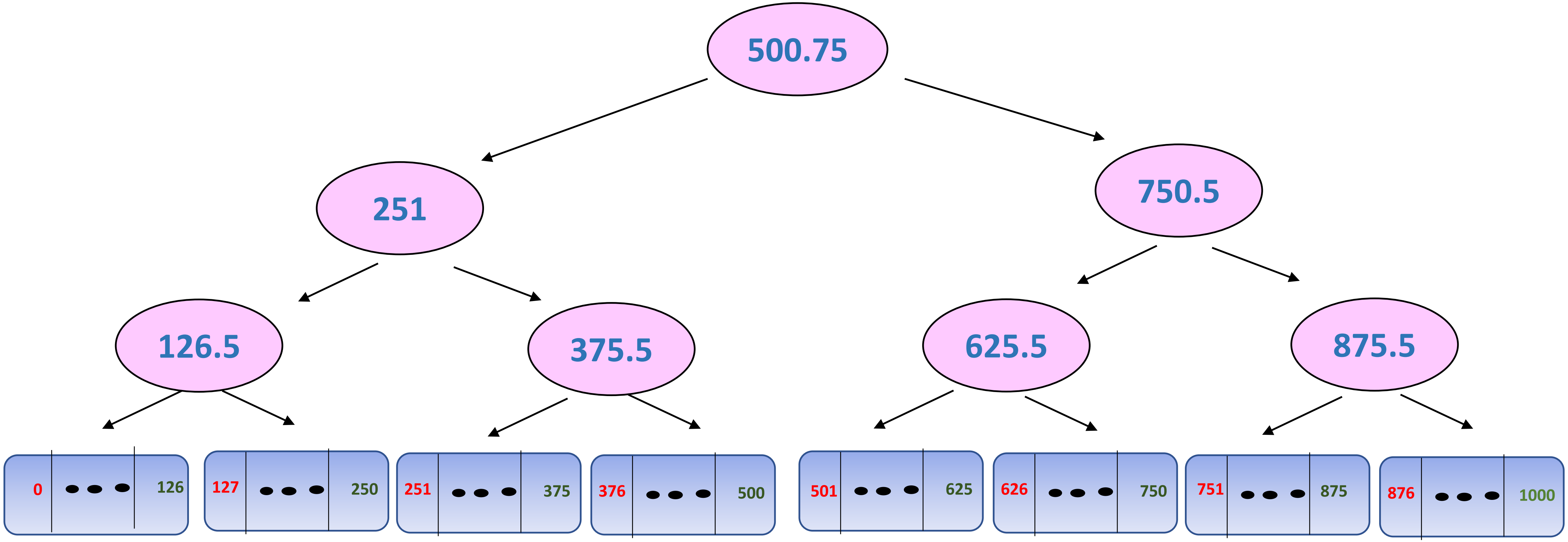



```
FIFOQueue<Node<T>>
```

```
FifoQueue<Node<T>> q = new FifoQueue<T>();
nodes.forEach(q::enqueue); // Assume that all nodes are in some Iterable structure
while(q.size() > 2){
    Node n1 = q.dequeue(), n2 = q.dequeue();
    q.enqueue(new Node((n1.key + n2.key) / 2, n1, n2));
} // At this point we only have two subtrees and have to connect them to a root node (q.size() == 2 is an invariant)
Node n1 = q.dequeue(), n2 = q.dequeue();
Node root = new Node((n1.key + n2.key) / 2, n1, n2));
```

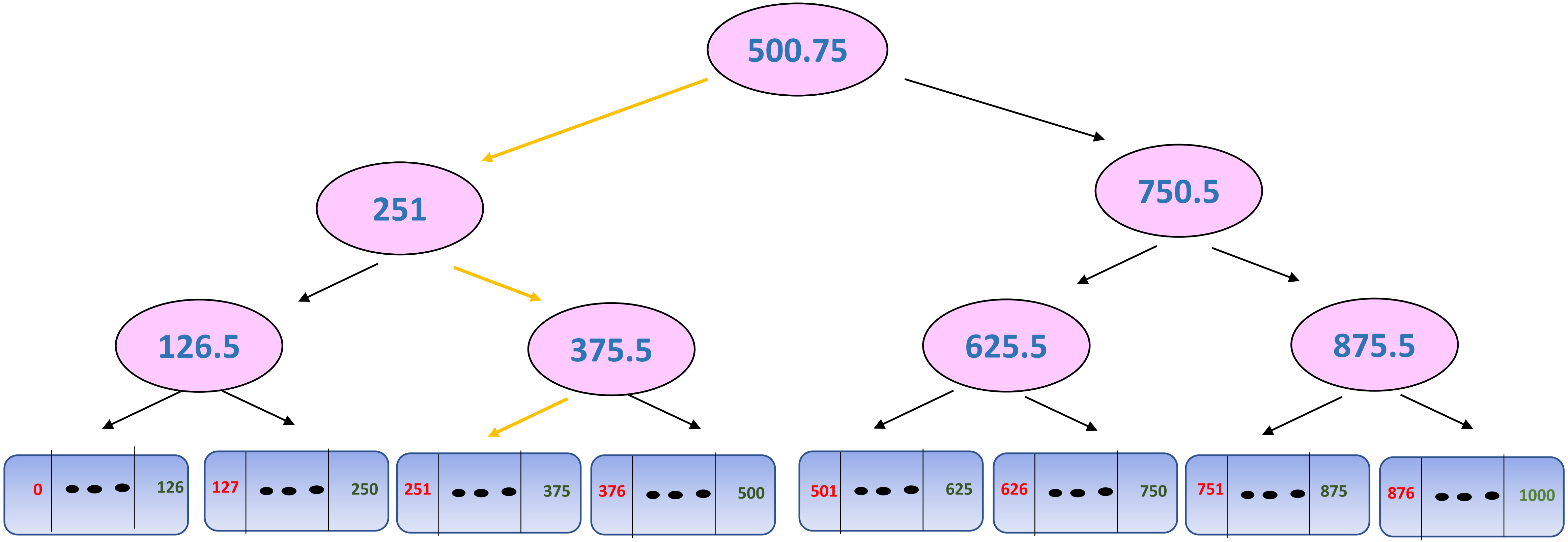
Using our index

```
SELECT * from STUDENTS WHERE ID > 250 AND ID <= 700
```



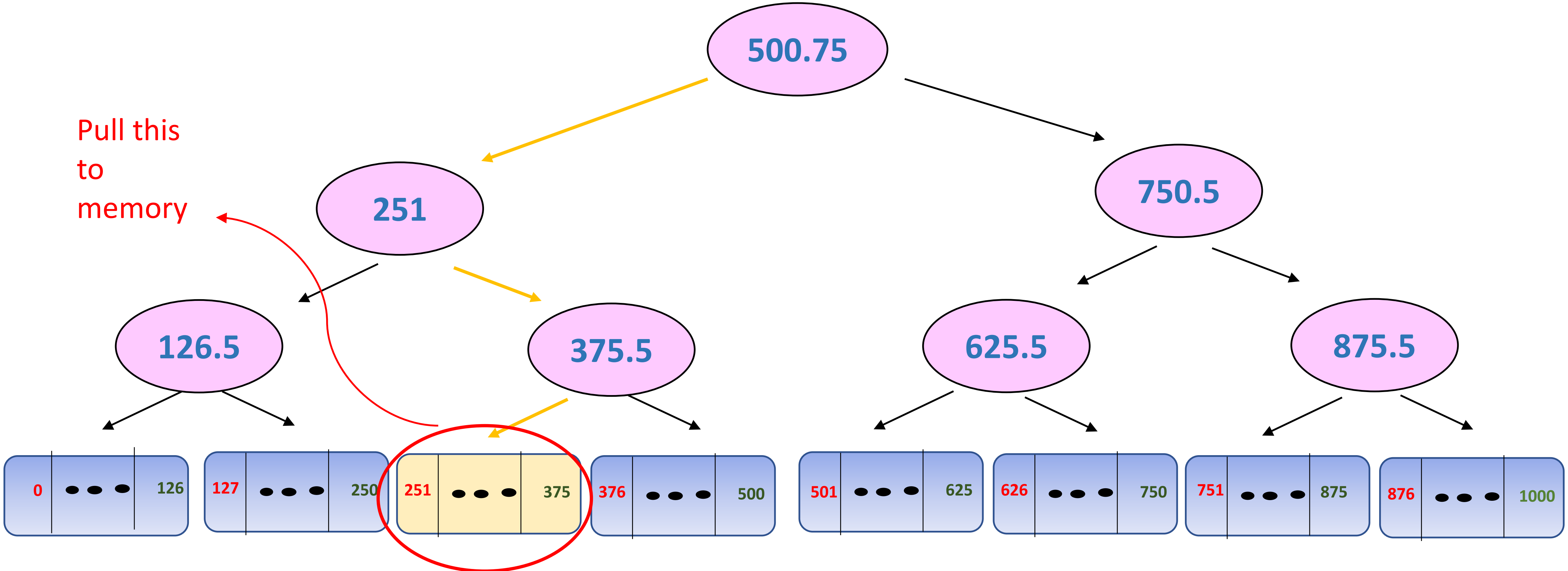
Using our index

```
SELECT * from STUDENTS WHERE ID > 250 AND ID <= 700
```



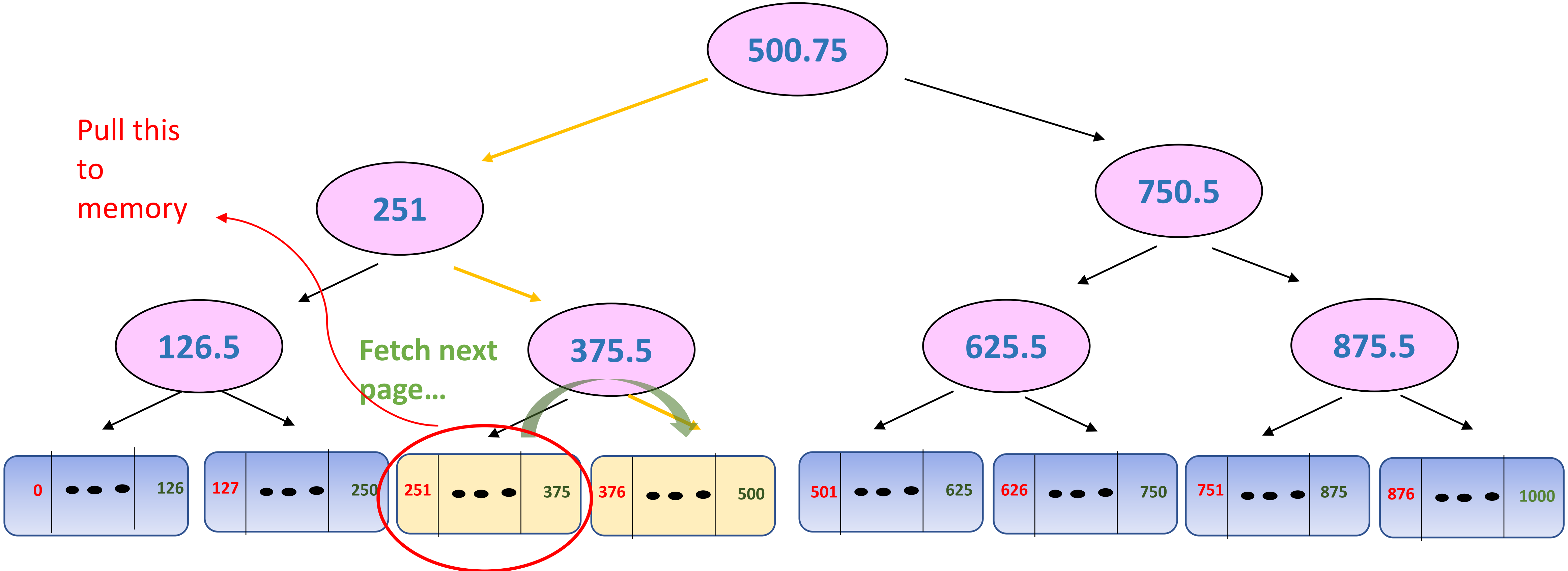
Using our index

```
SELECT * from STUDENTS WHERE ID > 250 AND ID <= 700
```



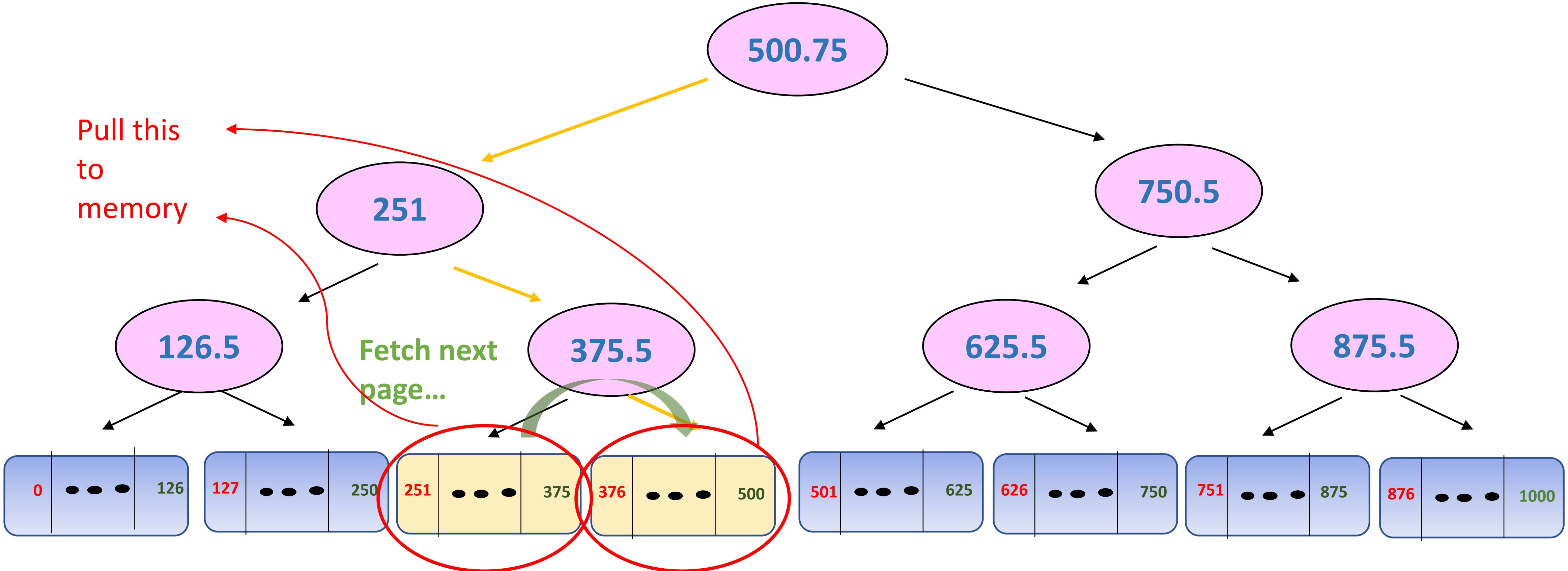
Using our index

```
SELECT * from STUDENTS WHERE ID > 250 AND ID <= 700
```



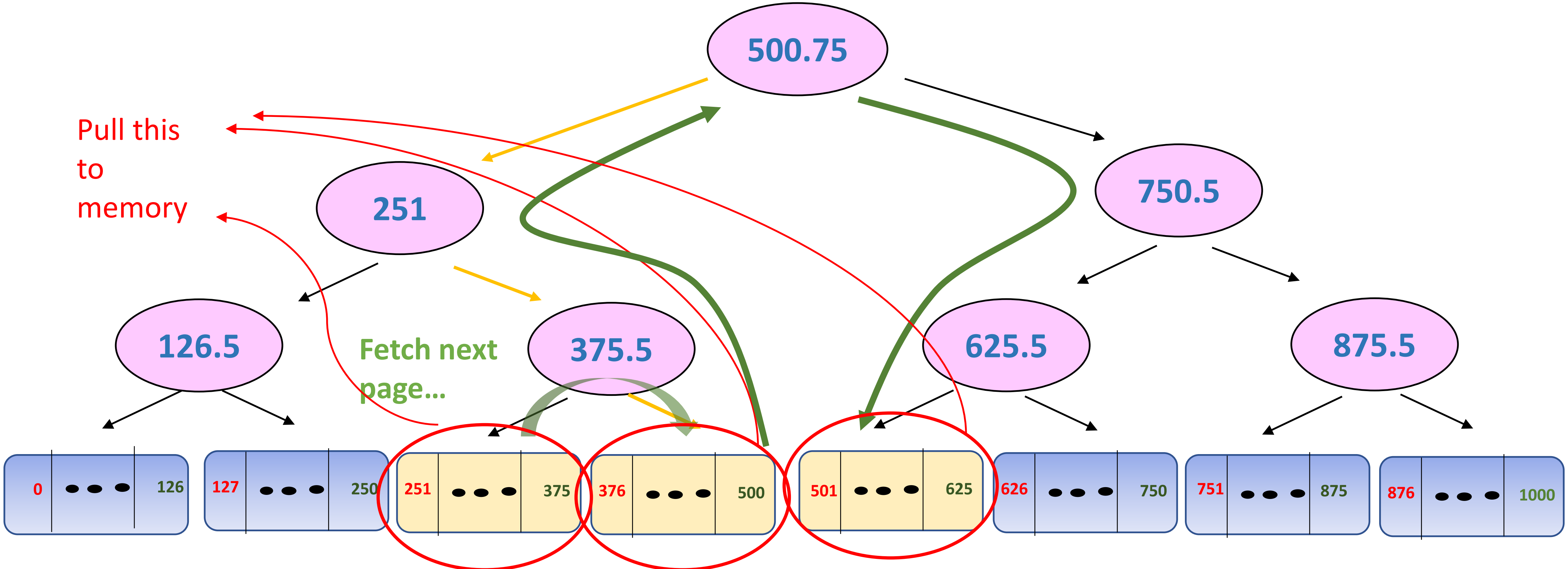
Using our index

```
SELECT * from STUDENTS WHERE ID > 250 AND ID <= 700
```



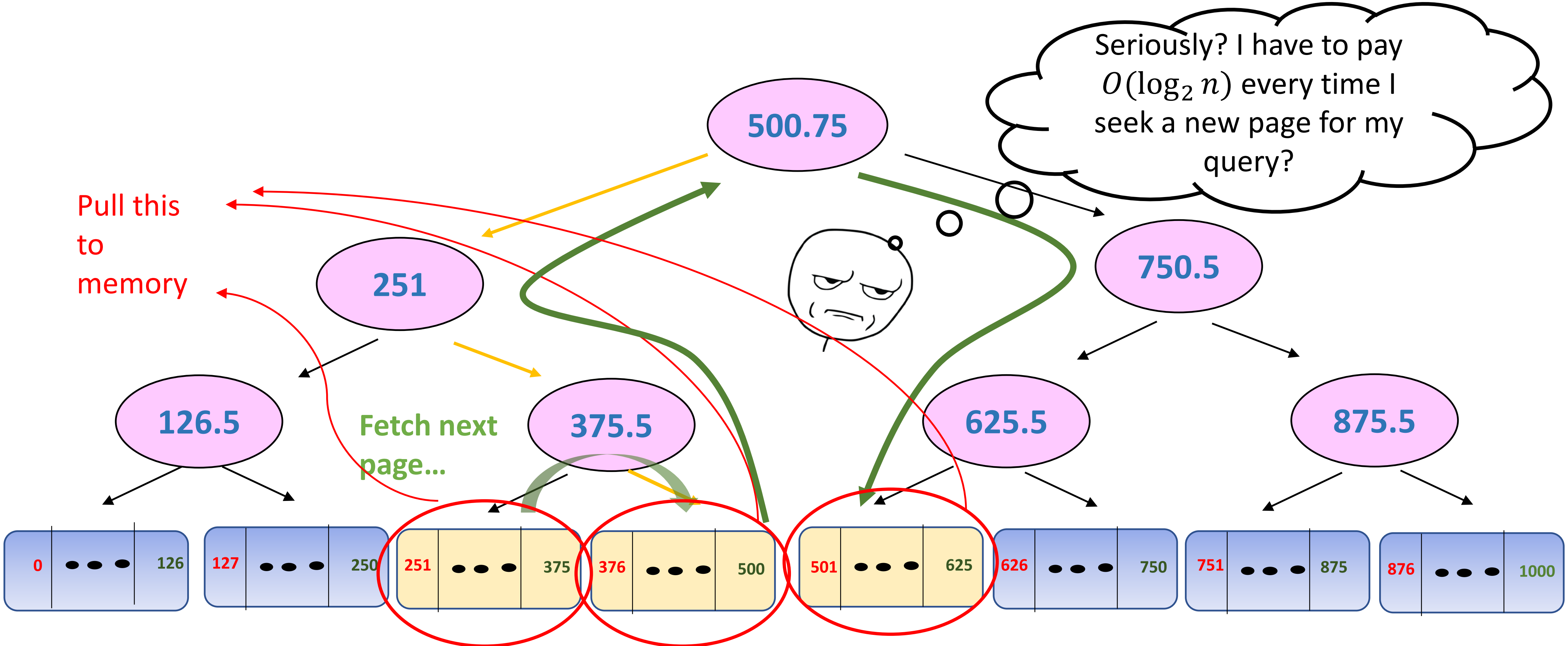
Using our index

```
SELECT * from STUDENTS WHERE ID > 250 AND ID <= 700
```



Using our index

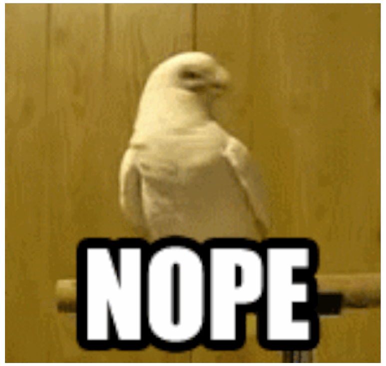
```
SELECT * from STUDENTS WHERE ID > 250 AND ID <= 700
```

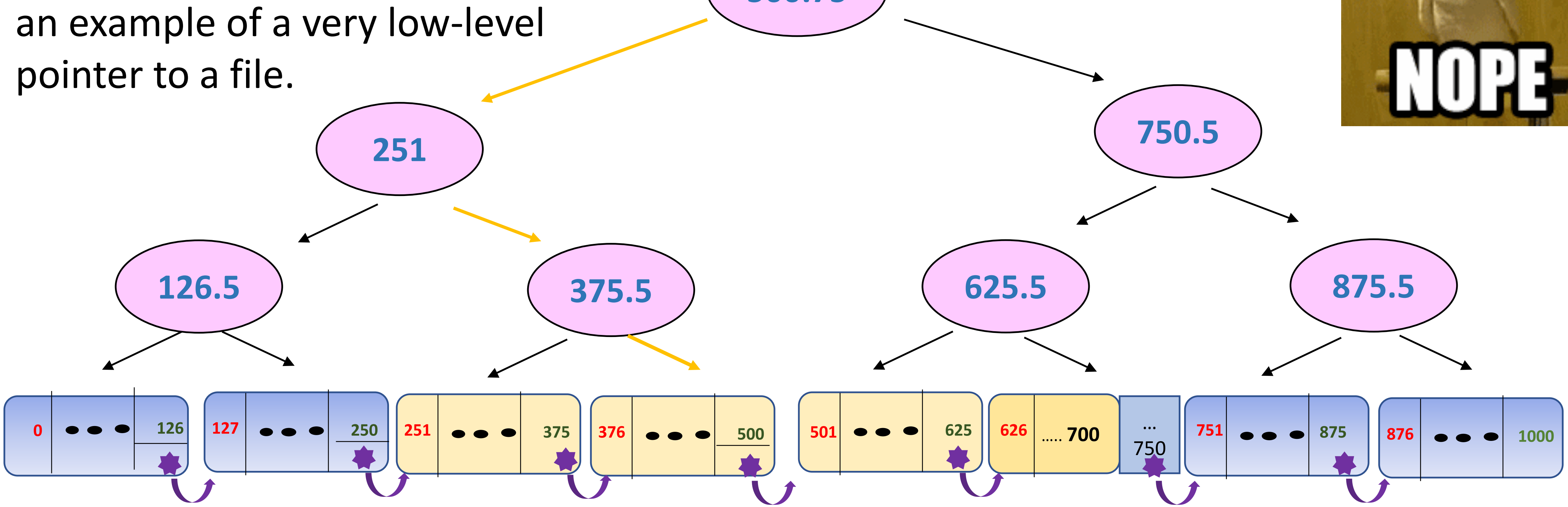


Using our index

```
SELECT * from STUDENTS WHERE ID > 250 AND ID <= 700
```

- **Store pointers on disk!**
- Look up C's `FILE*`("File-star") for an example of a very low-level pointer to a file.





Space Analysis (Main Memory)

- We have 7 nodes with 2 references and 1 float each.
  - 7 * 16 + 28 = 140 bytes total.
- Total index size: 140 bytes $= 0.4375\%$ of database size.
- 0.4375% sounds quite good, but what happens when our database is 128 GB in size? ☹
  - Then, 0.525% of 128GB is **672MB**.
  - **We can do better ☺**

Example of our new index

- Let's assume $p = 4$ (fanout of nodes = 4).
  - Non-root nodes hold between $\left\lceil \frac{p}{2} \right\rceil - 1 = 1$ and $p - 1 = 3$ keys.
  - Root can have between 1 and 3 keys.

Example of our new index

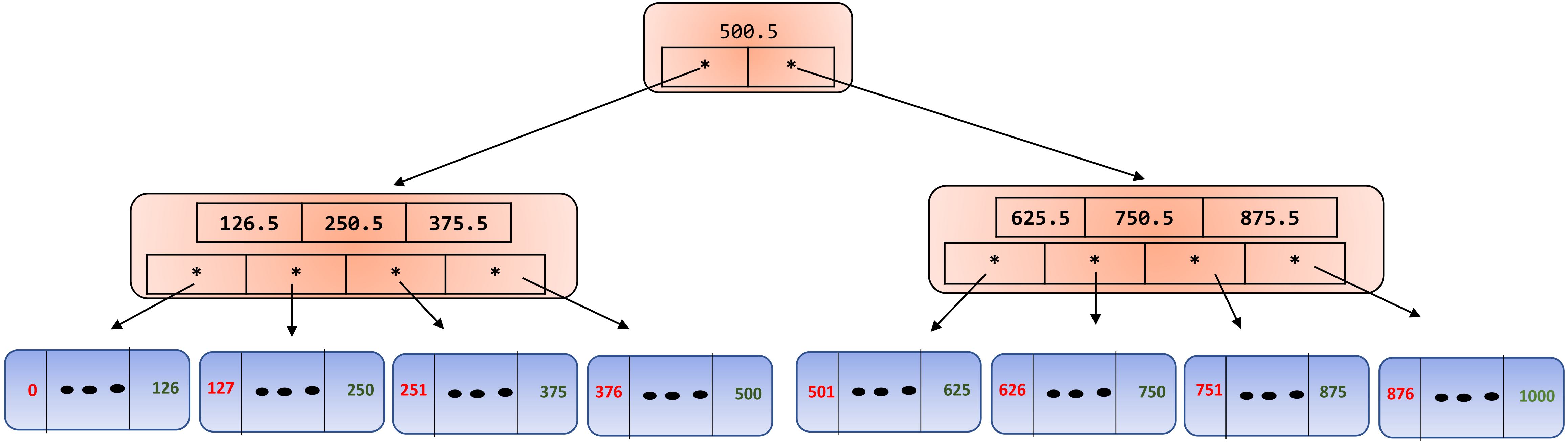
- Then, we can build the following index over the same data:

Example of our new index

- Then, we can build the following index over the same data:

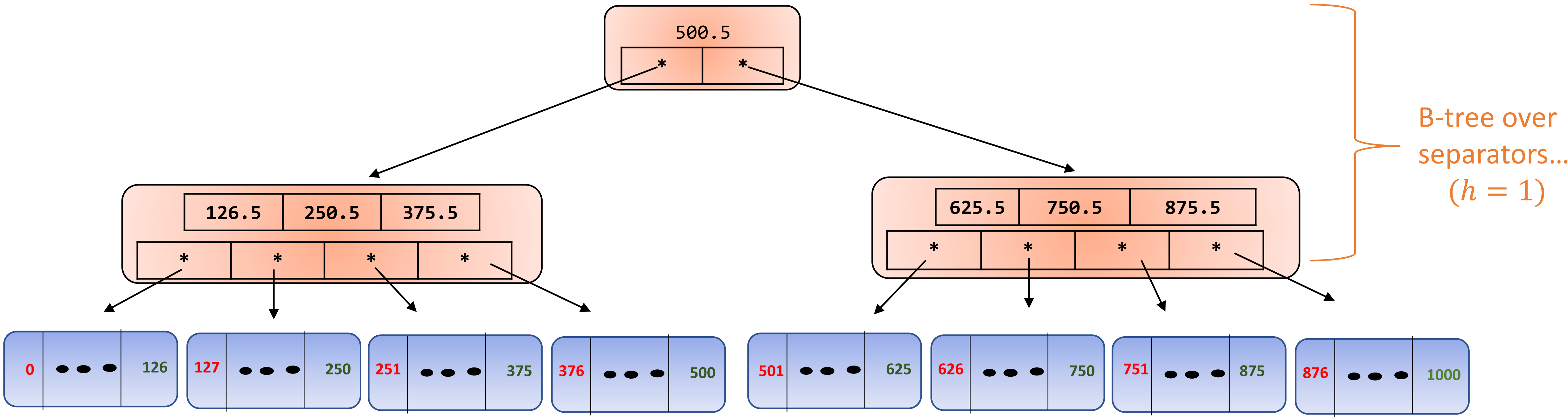
Example of our new index

- Then, we can build the following index over the same data:



Example of our new index

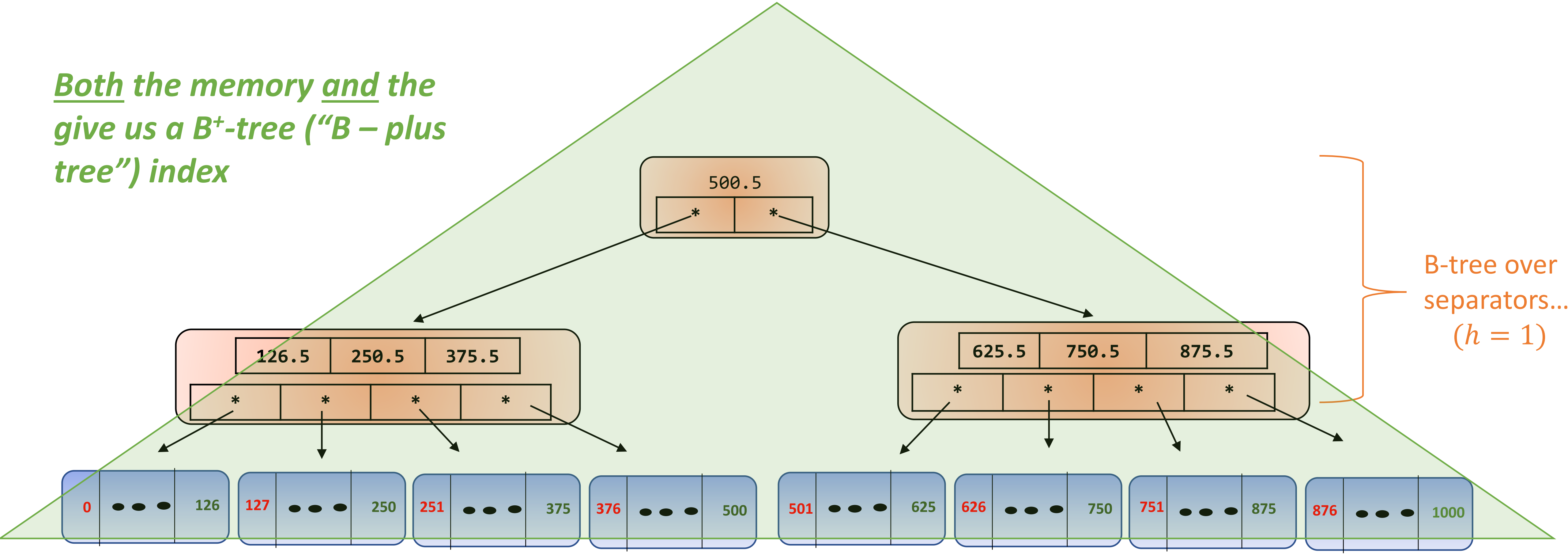
- Then, we can build the following index over the same data:



Example of our new index

- Then, we can build the following index over the same data:

Both the memory and the give us a B+-tree ("B – plus tree") index



Comparison to BST index

- Spatial Cost of B^+- Tree index = 2 * leaf_node_cost + root_cost
- Leaf_node_cost = 3 * `sizeof(float)` + 4 *`sizeof(ref)` = 3 * 4 + 4 * 8 = 44 bytes
- root_cost = sizeof(float) + 2 * sizeof(ref) = 4 + 16 = 20 bytes
- So Spatial Cost = 2 * 44 + 20 = 104 bytes < 140 bytes which was the cost for the BST index!
  - Wait till you see the spatial benefit in larger databases…

The power of B^+-trees

- Suppose my page size is 4KB (standard for most commercial PCs)

The power of B^+-trees

- Suppose my page size is 4KB (standard for most commercial PCs)
- Then, how large of a database can I index into with a B^+- tree of height $h = 2$ and $p = 8$?

The power of B+-trees

- Suppose my page size is 4KB (standard for most commercial PCs)
- Then, how large of a database can I index into with a B+- tree of height $h = 2$ and $p = 8$?

The B-Tree component…

The power of B+-trees

- Suppose my page size is 4KB (standard for most commercial PCs)
- Then, how large of a database can I index into with a B+- tree of height $h = 2$ and $p = 8$?

$\approx 128KB$

$\approx 256KB$

$\approx 0.5MB$

$\approx 2\ MB$

The power of B$^+$-trees

- Suppose my page size is 4KB (standard for most commercial PCs)
- Then, how large of a database can I index into with a B$^+$- tree of height $h = 2$ and $p = 8$?

$\approx 128KB$

$\approx 256KB$

$\approx 0.5MB$

$\approx 2\ MB$

1. #leaves in our B+-tree: $p^h = 64$
2. Each of them points to 8 pages
3. So $8 \times 64 = 8^3 = 512 = 2^9$ pages total
4. Therefore, $DB_{size} = 2^9 * 2^2\ KB = 2MB$

The power of B⁺-trees

- Suppose my page size is 4KB (standard for most commercial PCs)
- Then, how large of a database can I index into with a B⁺- tree of height $h = 2$ and $p = 8$?

$\approx 128KB$

$\approx 256KB$

$\approx 0.5MB$

$\approx 2\ MB$

1. #leaves in our B+-tree: $p^h = 64$
2. Each of them points to 8 pages
3. So $8 \times 64 = 8^3 = 512 = 2^9$ pages total
4. Therefore, $DB_{size} = 2^9 * 2^2\ KB = 2MB$

While our B+-trees spatial cost is:

$(8 * 8 + 7 * 4)(1 + 8 + 8^2) =$ $6717\text{B} = 6.717KB \approx$ 0.336% of DB_{size}!

Similar exercise

- Page size = 4KB
- With a height $h = 3$ and a fanout of $p = 128$, how large of a database can I index into?

Similar exercise

- Page size = 4KB
- With a height $h = 3$ and a fanout of $p = 128$, how large of a database can I index into?

- $p^h = (2^7)^3 = 2^{21}$ leaves.
- $2^{21} * 128 = 2^{28}$ pages
- $2^{28} * 4KB = 2^{30}KB \approx 1073.74\ GB \approx 1.073\ TB$

Similar exercise

- Page size = 4KB
- With a height $h = 3$ and a fanout of $p = 128$, how large of a database can I index into?

- $p^h = (2^7)^3 = 2^{21}$ leaves.
- $2^{21} * 128 = 2^{28}$ pages
- $2^{28} * 4KB = 2^{30}KB \approx 1073.74\ GB \approx 1.073\ TB$

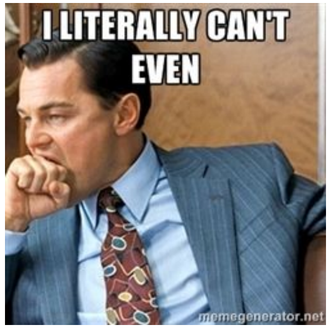
- #nodes = $1 + 128 + 128^2 + 128^3 = \frac{128^4-1}{128-1} = \mathbf{2113665}$
- Since $p = 128$,
  - $127 * 4 \approx .5\text{KB}$
  - $128 * 8 = 1\text{KB}$
- So, total cost =$2113665 * 1.5KB = 3170497.5KB \approx 3.1GB$

Similar exercise

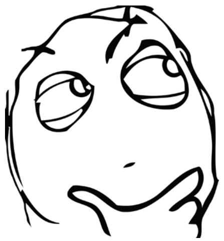
- Page size = 4KB
- With a height $h = 3$ and a fanout of $p = 128$, how large of a database can I index into?

- $p^h = (2^7)^3 = 2^{21}$ leaves.
- $2^{21} * 128 = 2^{28}$ pages
- $2^{28} * 4KB = 2^{30}KB \approx 1073.74\ GB \approx 1.073\ TB$

- #nodes = $1 + 128 + 128^2 + 128^3 = \frac{128^4 - 1}{128 - 1} = \mathbf{2113665}$
- Since $p = 128$,
  - $127 * 4 \approx .5\text{KB}$
  - $128 * 8 = 1\text{KB}$
- So, total cost =$2113665 * 1.5KB = 3170497.5KB \approx 3.1GB$

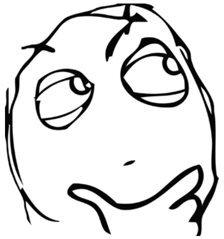


0.289% of DB SIZE!!



Thought experiment

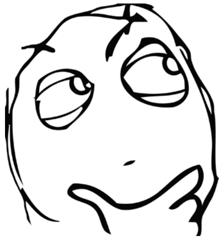
- For the same size DB, what would the size of a BST-based index be?



Thought experiment

- For the same size DB, what would the size of a BST-based index be?

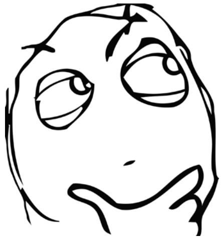
- **(DB size = $2^{30}KB$)** $\Longrightarrow \#pages = \frac{2^{30}\cancel{KB}}{2^{2}\cancel{KB}} = 2^{28}$



Thought experiment

- For the same size DB, what would the size of a BST-based index be?

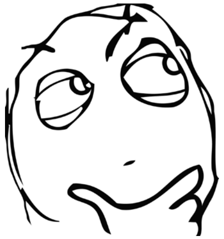
- **(DB size =** $2^{30}KB$**)** $\Longrightarrow \#pages = \frac{2^{30}\cancel{KB}}{2^{2}\cancel{KB}} = 2^{28}$
- Every **pair** of pages connected by a leaf node, so #leaf nodes $= \frac{2^{28}}{2} = 2^{27}$



Thought experiment

- For the same size DB, what would the size of a BST-based index be?

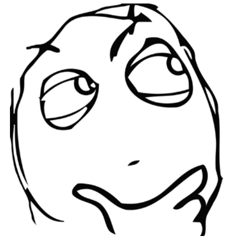
- (DB size = $2^{30}KB$) $\Longrightarrow \#pages = \frac{2^{30}\cancel{KB}}{2^{2}\cancel{KB}} = 2^{28}$
- Every **pair** of pages connected by a leaf node, so #leaf nodes $= \frac{2^{28}}{2} = 2^{27}$
- Since leaf nodes are 2^{27}, height = 27 *(28 levels in the tree)*



Thought experiment

- For the same size DB, what would the size of a BST-based index be?

- (DB size = $2^{30}KB$) $\Longrightarrow \#pages = \frac{2^{30}\cancel{KB}}{2^{2}\cancel{KB}} = 2^{28}$
- Every **pair** of pages connected by a leaf node, so #leaf nodes $= \frac{2^{28}}{2} = 2^{27}$
- Since leaf nodes are 2^{27}, height = 27 *(28 levels in the tree)*
- #nodes = $2^{0} + 2^{1} + \cdots 2^{27} = \frac{2^{28}-1}{2-1} = 268{,}435{,}455$



Thought experiment

- For the same size DB, what would the size of a BST-based index be?

- (DB size = $2^{30}KB$) $\Longrightarrow \#pages = \frac{2^{30}\cancel{KB}}{2^{2}\cancel{KB}} = 2^{28}$
- Every **pair** of pages connected by a leaf node, so #leaf nodes $= \frac{2^{28}}{2} = 2^{27}$
- Since leaf nodes are 2^{27}, height = 27 *(28 levels in the tree)*
- #nodes = $2^0 + 2^1 + \cdots 2^{27} = \frac{2^{28}-1}{2-1} = 268{,}435{,}455$
- node_cost = 2*`sizeof(ref)` + `sizeof(float)` = 20 bytes

Thought experiment

- For the same size DB, what would the size of a BST-based index be?

- **(DB size =** $2^{30}KB$**)** $\Longrightarrow \#pages = \frac{2^{30}\cancel{KB}}{2^{2}\cancel{KB}} = 2^{28}$
- Every **pair** of pages connected by a leaf node, so #leaf nodes = $\frac{2^{28}}{2} = 2^{27}$
- Since leaf nodes are 2^{27}, height = 27 *(28 levels in the tree)*
- #nodes = $2^{0} + 2^{1} + \cdots 2^{27} = \frac{2^{28}-1}{2-1} = 268{,}435{,}455$
- node_cost = 2*`sizeof(ref)` + `sizeof(float)` = 20 bytes

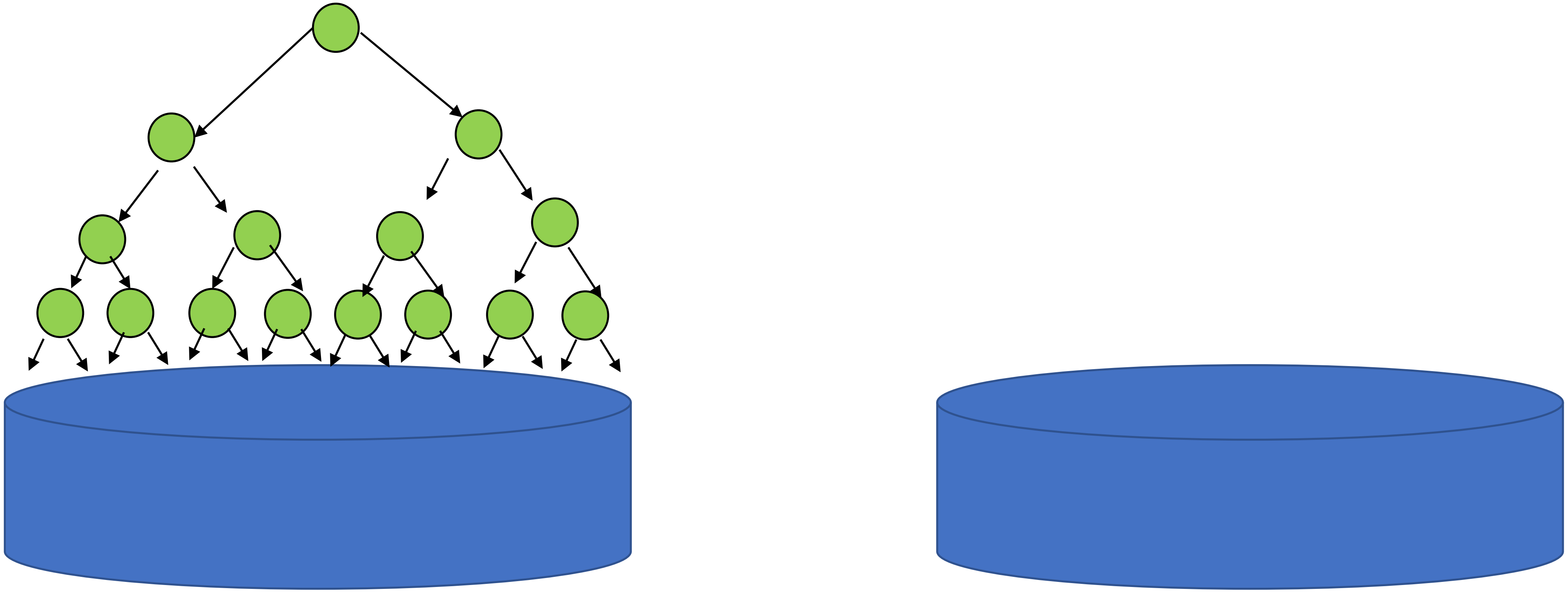
Total cost= **5,368,709,100 bytes = 5.3687091 GB** $\approx$ **0.502% of DB Size!**

So why does this happen?

- **Mathematical reason #1:** Because you have many leaves ℓ, and a small value of $p = 2$, so you need a large value for h to satisfy $\ell = p^h$!
  - This leads to a large height for the tree and an increased number of (large) summands in the sum of the geometric progression!
- **Mathematical reason #2:** Because the sum of the geometric progression when the value of h is large has many more (exponential) terms!
- **Intuitively**: Because, to cover a large DB, binary trees are **too tall for their own good**: a significant number of their subtrees can be collapsed into B-Tree nodes with appropriate separators.
  - Several subtrees are thus **redundant** and add to an **intractable storage cost for the index.**

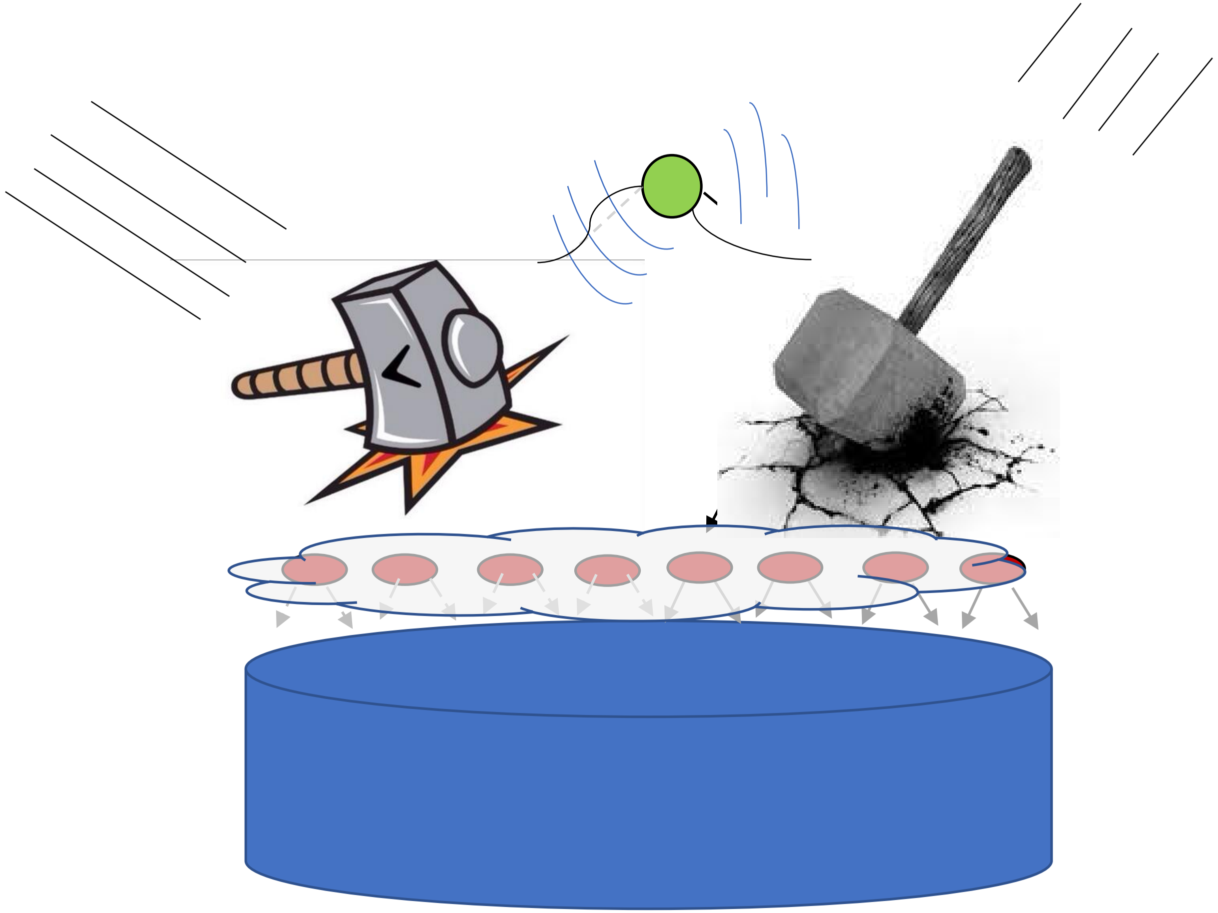
Mnemonic rule

- When indexing into a large DB….

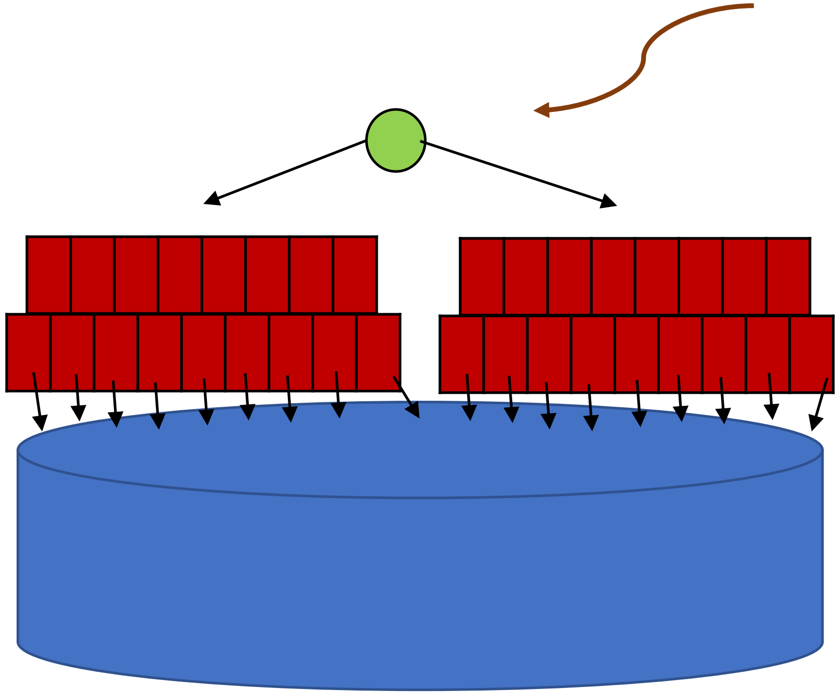


Mnemonic rule

- When indexing into a large DB….



Go short and fat!



What about different page sizes?

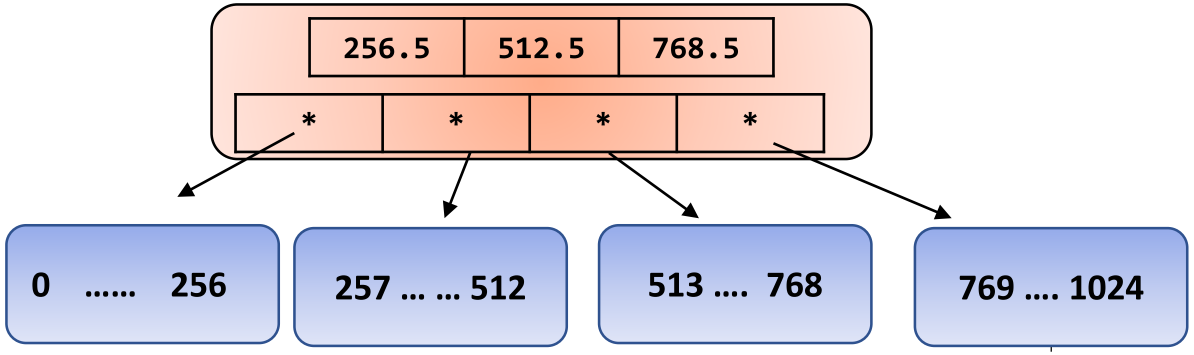
- Oracle's UltraSPARC07 arc defines 8KBs of page size.
- *How does that affect the binary tree index's size?*
- Again, original size of data = 2^{30}KB
- Since page size is 8KB, there are $2^{30}/_8 = 2^{27}$ pages
- So we need 2^{26} leaves, for a tree of height 26 (just one less… ☹)
  - So we're not helped much ☹

What about different page sizes?

- What about the B-Tree of fanout $p = 4$?
- #pages = $\frac{1000*32\ B}{8*1000\ B} = 4$

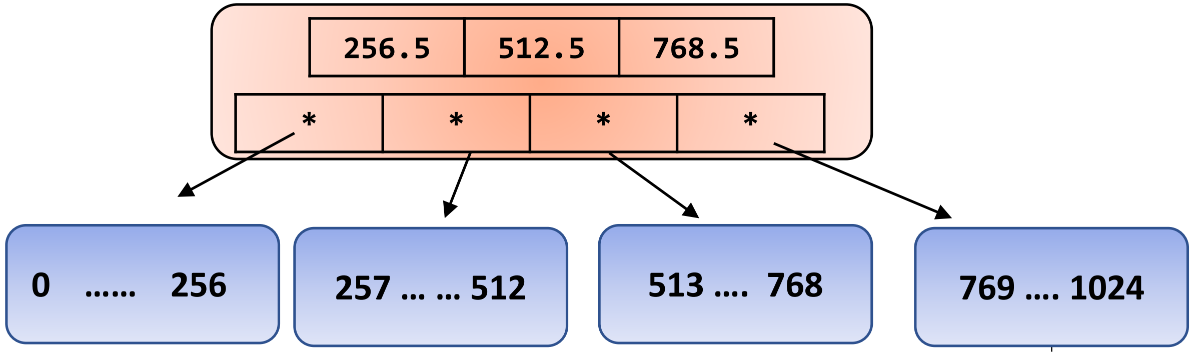
What about different page sizes?

- What about the B-Tree of fanout $p = 4$?
- #pages = $\frac{1000*32\ B}{8*1000\ B} = 4$
- A single root node is needed! :O



What about different page sizes?

- What about the B-Tree of fanout $p = 4$?
- #pages = $\frac{1000*32\ B}{8*1000\ B} = 4$
- A single root node is needed! :O



Memory cost = 3 * `sizeof(float)` + 4*`sizeof(ref)` = 44 bytes = 0.1375% of DB size

Deciding on fan-out and height

- Suppose that we have a database D of size $|D| = 4\ GB$, stored in a drive with page size $PS = 4KB$.
- To index D, we want to use a B+-tree with $p = 8$.

Deciding on fan-out and height

- Suppose that we have a database D of size $|D| = 4\ GB$, stored in a drive with page size $PS = 4KB$.
- To index D, we want to use a B+-tree with $p = 8$.
- Q1: What should its height be in order for it to index the entirety of D?

Deciding on fan-out and height

- Suppose that we have a database D of size $|D| = 4\ GB$, stored in a drive with page size $PS = 4KB$.
- To index D, we want to use a B+-tree with $p = 8$.
- Q1: What should its height be in order for it to index the entirety of D?
- #pages $pg = \frac{4 \times 10^6 KB}{4\ KB} = 10^6$

Deciding on fan-out and height

- Suppose that we have a database D of size $|D| = 4\ GB$, stored in a drive with page size $PS = 4KB$.
- To index D, we want to use a B+-tree with $p = 8$.
- Q1: What should its height be in order for it to index the entirety of D?
- #pages $pg = \frac{4 \times 10^6 KB}{4\ KB} = 10^6$
- So #leaves $\ell = \frac{10^6}{p} = 125\ * 10^3$

Deciding on fan-out and height

- Suppose that we have a database D of size $|D| = 4\ GB$, stored in a drive with page size $PS = 4KB$.
- To index D, we want to use a B+-tree with $p = 8$.
- Q1: What should its height be in order for it to index the entirety of D?
- #pages $pg = \frac{4\times 10^6 KB}{4\ KB} = 10^6$
- So #leaves $\ell = \frac{10^6}{p} = 125\ * 10^3$
- But! $\ell = p^h \Rightarrow h = \log_p \ell = \log_8 142857 = 5.7 \Rightarrow h = 6$

Spatial cost of this index

- Q2: What's the spatial cost of this index?

Spatial cost of this index

- Q2: What's the spatial cost of this index?

- Node cost = 4*7 + 8 * 8 =92 bytes

Spatial cost of this index

- Q2: What's the spatial cost of this index?

- Node cost = 4*7 + 8 * 8 =92 bytes
- Total cost = 92 $\times (1 + 8 + \ldots + 8^6)$

Spatial cost of this index

- Q2: What's the spatial cost of this index?

- Node cost = 4*7 + 8 * 8 =92 bytes
- Total cost = 92 $\times \underbrace{(1 + 8 + \ldots + 8^6)}_{\sum_{i=0}^{6} 8^i} = 92 \times \frac{8^7-1}{8-1} \approx 27MB$

Spatial cost of this index

- Q2: What's the spatial cost of this index?

- Node cost = 4*7 + 8 * 8 =92 bytes
- Total cost = 92 × $\underbrace{(1 + 8 + \ldots + 8^6)}_{\sum_{i=0}^{6} 8^i} = 92 \times \frac{8^7-1}{8-1} \approx 27MB$

0.675% of DB_Size!

Formulae

- Given the size of the database, D, **<u>in GB</u>**, and the page size PS **<u>in KB</u>**, we have #pages pg:

$$pg \leftarrow \frac{D \times 10^6}{PS} \quad (1)$$

- There are p pages pointed to by a leaf, so the #leaves ℓ:

$$\ell \leftarrow \frac{pg}{p} \overset{(1)}{=} \frac{D \times 10^6}{PS * p} \quad (2)$$

Formulae

- But since the tree is built bottom-up, it is a "perfect" p-tree, so we know that

$$\ell = p^h \qquad (3)$$

- Combining (3) with (2) allows us to connect the crucial parameters p and h in one formula:

$$p^h = \frac{D \times 10^6}{PS * p}$$

Formulae

- But since the tree is built bottom-up, it is a "perfect" p-tree, so we know that

$$\ell = p^h \qquad (3)$$

- Combining (3) with (2) allows us to connect the crucial parameters p and h in one formula:

$$p^h = \frac{D \times 10^6}{PS * p} \Leftrightarrow p^{h+1} = \frac{D \times 10^6}{PS}$$

Formulae

- But since the tree is built bottom-up, it is a "perfect" p-tree, so we know that

$$\ell = p^h \qquad (3)$$

- Combining (3) with (2) allows us to connect the crucial parameters p and h in one formula:

$$p^{h+1} = \frac{D \times 10^6}{PS}$$

Given p

$$\boxed{h = \log_p \frac{D \times 10^6}{PS} - 1}$$

Given h

$$\boxed{p = \sqrt[h+1]{\frac{D \times 10^6}{PS}}}$$

Formulae

- Given both p and h and assuming 4-byte separators and 8-byte references, we have:

$$Cost_{SPACE} = (12p - 4) \cdot \frac{p^{h+1} - 1}{p - 1}$$

Formulae

- Given both p and h and assuming 4-byte separators and 8-byte references, we have:

$$Cost_{SPACE} = \underbrace{(12p - 4)}_{\text{Cost of node}} \cdot \underbrace{\frac{p^{h+1} - 1}{p - 1}}_{\text{Number of nodes}}$$

Cost of node:
$12p - 4 = 4(p - 1) + 8p$

Number of nodes:
Sum of geometric progression

Applying our formulae

- $p^{h+1} = \frac{D \times 10^6}{PS}$ (identity that connects p and h)
- $Cost_{SPACE} = (12p - 4) \cdot \frac{p^{h+1} - 1}{p - 1}$

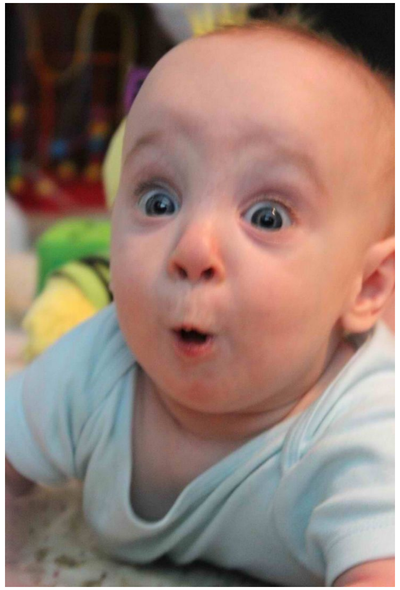
Applying our formulae

- $p^{h+1} = \frac{D \times 10^6}{PS}$ (identity that connects p and h)
- $Cost_{SPACE} = (12p - 4) \cdot \frac{p^{h+1} - 1}{p - 1}$

- Let $D = 1TB$ and $PS = 4KB$. Suppose we want $h = 3$. Then,

$p^4 = \frac{D \times 10^6}{PS} \Rightarrow p^4 = \frac{10^9}{4} = 250 * 10^6 \Rightarrow p \approx 126$ and our spatial cost is $(12 * 126 - 4) * \frac{126^4 - 1}{126 - 1} = 3{,}040{,}699{,}532$ bytes $\approx 3.04\text{GB}$

Applying our formulae



- $p^{h+1} = \frac{D \times 10^6}{PS}$ (identity that connects p and h)
- $Cost_{SPACE} = (12p - 4) \cdot \frac{p^{h+1} - 1}{p - 1}$

- Let $D = 1TB$ and $PS = 4KB$. Suppose we want $h = 3$. Then,

$p^4 = \frac{D \times 10^6}{PS} \Rightarrow p^4 = \frac{10^9}{4} = 250 * 10^6 \Rightarrow p \approx 126$ and our spatial cost is $(12 * 126 - 4) * \frac{126^4 - 1}{126 - 1} = 3{,}040{,}699{,}532$ bytes ≈ 3.04GB=0.304% of DB size!

Applying our formulae

- $p^{h+1} = \frac{D \times 10^6}{PS}$ (identity that connects p and h)
- $Cost_{SPACE} = (12p - 4) \cdot \frac{p^{h+1} - 1}{p - 1}$

- Let $D = 1TB$ and $PS = 4KB$. Suppose now we want $p = 64$.

Applying our formulae

- $p^{h+1} = \frac{D \times 10^6}{PS}$ (identity that connects p and h)
- $Cost_{SPACE} = (12p - 4) \cdot \frac{p^{h+1} - 1}{p - 1}$

- Let $D = 1TB$ and $PS = 4KB$. Suppose now we want $p = 64$. Then,

$h = \log_{64}(250 * 10^6) - 1 \approx 3.65 \Rightarrow h = 4$ and our spatial cost is $(12 * 64 - 4) * \frac{64^5 - 1}{64 - 1} = 13{,}021{,}250{,}044 \approx 13.021\text{GB} = 1.3\%$ of DB Size ☹

Applying our formulae

- $p^{h+1} = \frac{D \times 10^6}{PS}$ (identity that connects p and h)
- $Cost_{SPACE} = (12p - 4) \cdot \frac{p^{h+1}-1}{p-1}$

- Let $D = 1TB$ and $PS = 4KB$. Suppose now we want $p = 64$. Then,

$h = \log_{64}(250 * 10^6) - 1 \approx 3.65 \Rightarrow h = 4$ and our spatial cost is $(12 * 64 - 4) * \frac{64^5-1}{64-1} = 13{,}021{,}250{,}044 \approx 13.021\text{GB} = 1.3\%$ of DB Size ☹

(Expected, the height increased!)