



RED BLACK TREES

JINANG SHAH (B22CS027)
VISHESH SACHDEVA (B22AI050)
AMBATI RAHUL REDDY (B22CS088)

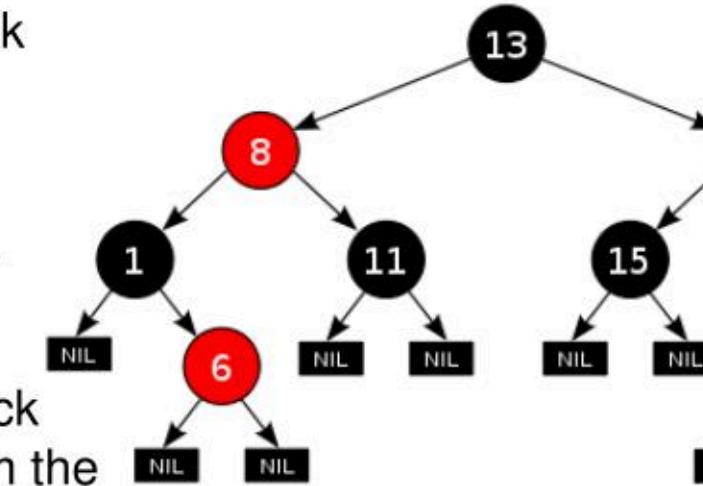
WHAT ARE RED BLACK TREES?

Red-Black tree is a binary search tree in which every node is colored with either red or black. It is a type of self balancing binary search tree. It has a good efficient worst case running time complexity.

Red Black Trees are a type of balanced binary search tree that use a specific set of rules to ensure that the tree is always balanced. This balance guarantees that the time complexity for operations such as insertion, deletion, and searching is always $O(\log n)$, regardless of the initial shape of the tree.

Rules of Red/Black Trees

1. Every **node** is either red or black
 - A **Node** is a non-null leaf
 - A **NIL** is a null leaf
2. The root node is always black
3. Every leaf (NIL) is black
4. If a node is red, then both its children are black
 - two red nodes may not be adjacent
 - But if a node is black, its children can be red or black
5. For each node, all paths from the node to descendant leaves contain the same number of black nodes



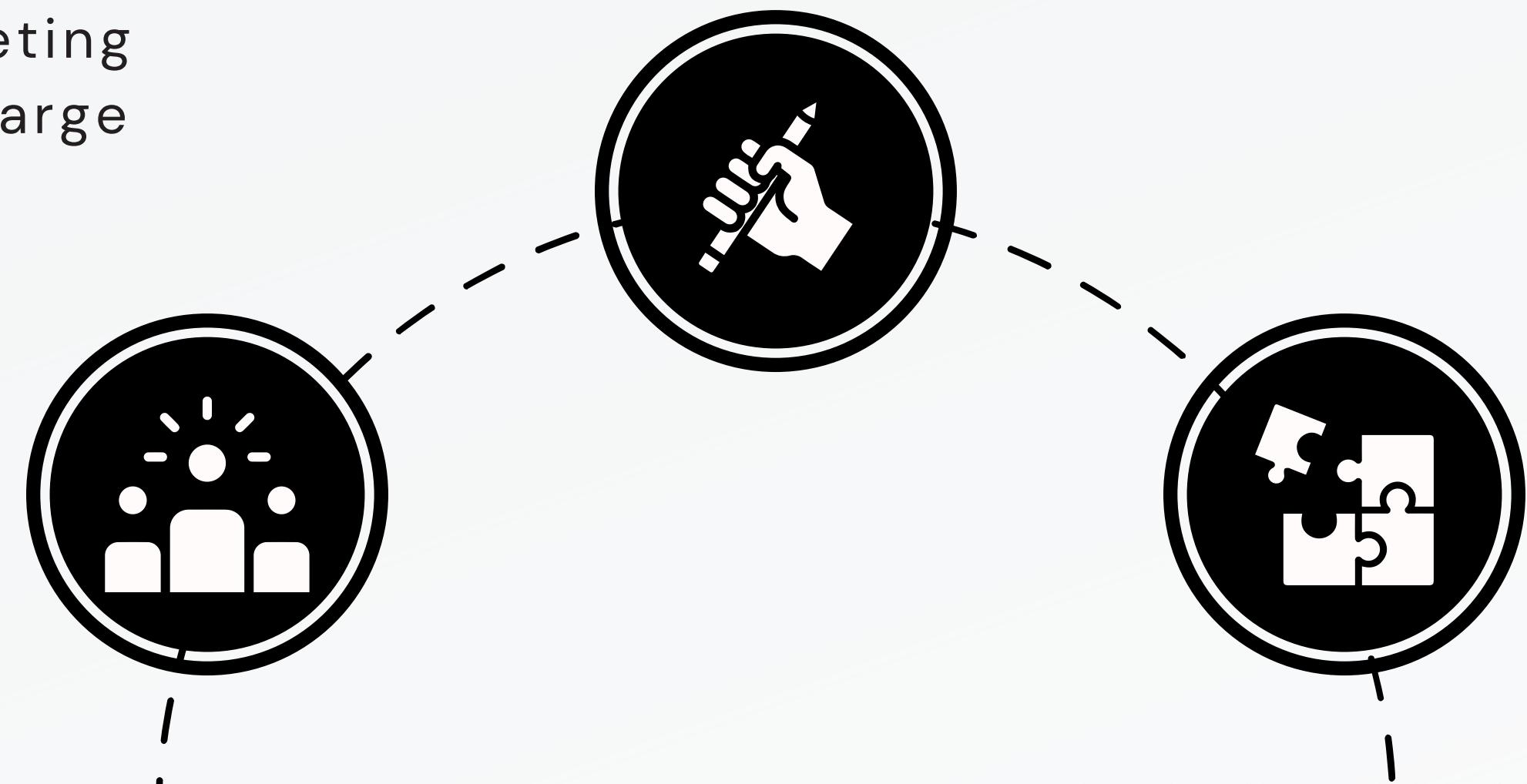
OBJECTIVES

Objective 1

We aim to implement
Red Black Trees
which can be used
for inserting, deleting
and searching a large
dataset.

Objective 2

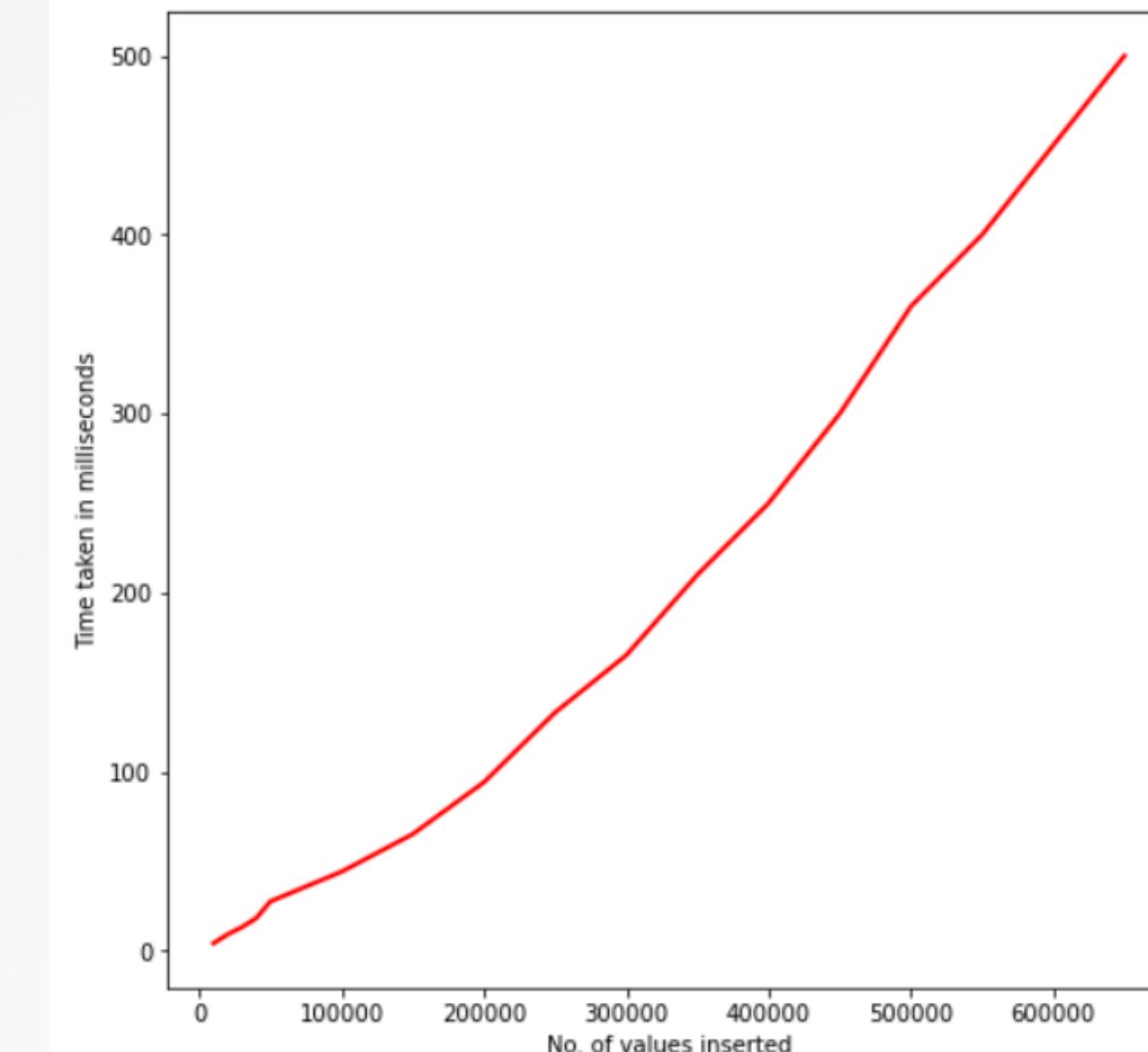
To Provide a fast
and efficient way
of searching in a
big datasets.



TIME COMPLEXITIES IN RB TREES

- Insertion, Deletion and Searching are performed in $O(\log n)$ time complexity.
- Inspite insertion of large amount of data single searching and deletion is performed in 0.044 sec and 0.002 sec respectively.

```
x = np.array([1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 21, 22, 23, 24, 25, 26, 27, 28, 29, 30, 31, 32, 33, 34, 35, 36, 37, 38, 39, 40, 41, 42, 43, 44, 45, 46, 47, 48, 49, 50, 51, 52, 53, 54, 55, 56, 57, 58, 59, 60, 61, 62, 63, 64, 65, 66, 67, 68, 69, 70, 71, 72, 73, 74, 75, 76, 77, 78, 79, 80, 81, 82, 83, 84, 85, 86, 87, 88, 89, 90, 91, 92, 93, 94, 95, 96, 97, 98, 99, 100, 101, 102, 103, 104, 105, 106, 107, 108, 109, 110, 111, 112, 113, 114, 115, 116, 117, 118, 119, 120, 121, 122, 123, 124, 125, 126, 127, 128, 129, 130, 131, 132, 133, 134, 135, 136, 137, 138, 139, 140, 141, 142, 143, 144, 145, 146, 147, 148, 149, 150, 151, 152, 153, 154, 155, 156, 157, 158, 159, 160, 161, 162, 163, 164, 165, 166, 167, 168, 169, 170, 171, 172, 173, 174, 175, 176, 177, 178, 179, 180, 181, 182, 183, 184, 185, 186, 187, 188, 189, 190, 191, 192, 193, 194, 195, 196, 197, 198, 199, 200, 201, 202, 203, 204, 205, 206, 207, 208, 209, 210, 211, 212, 213, 214, 215, 216, 217, 218, 219, 220, 221, 222, 223, 224, 225, 226, 227, 228, 229, 230, 231, 232, 233, 234, 235, 236, 237, 238, 239, 240, 241, 242, 243, 244, 245, 246, 247, 248, 249, 250, 251, 252, 253, 254, 255, 256, 257, 258, 259, 259, 260, 261, 262, 263, 264, 265, 266, 267, 268, 269, 270, 271, 272, 273, 274, 275, 276, 277, 278, 279, 280, 281, 282, 283, 284, 285, 286, 287, 288, 289, 290, 291, 292, 293, 294, 295, 296, 297, 298, 299, 299, 300, 301, 302, 303, 304, 305, 306, 307, 308, 309, 309, 310, 311, 312, 313, 314, 315, 316, 317, 318, 319, 319, 320, 321, 322, 323, 324, 325, 326, 327, 328, 329, 329, 330, 331, 332, 333, 334, 335, 336, 337, 338, 339, 339, 340, 341, 342, 343, 344, 345, 346, 347, 348, 349, 349, 350, 351, 352, 353, 354, 355, 356, 357, 358, 359, 359, 360, 361, 362, 363, 364, 365, 366, 367, 368, 369, 369, 370, 371, 372, 373, 374, 375, 376, 377, 378, 378, 379, 380, 381, 382, 383, 384, 385, 386, 387, 388, 389, 389, 390, 391, 392, 393, 394, 395, 396, 397, 398, 399, 399, 400, 401, 402, 403, 404, 405, 406, 407, 408, 409, 409, 410, 411, 412, 413, 414, 415, 416, 417, 418, 418, 419, 420, 421, 422, 423, 424, 425, 426, 427, 428, 429, 429, 430, 431, 432, 433, 434, 435, 436, 437, 438, 439, 439, 440, 441, 442, 443, 444, 445, 446, 447, 448, 449, 449, 450, 451, 452, 453, 454, 455, 456, 457, 458, 459, 459, 460, 461, 462, 463, 464, 465, 466, 467, 468, 469, 469, 470, 471, 472, 473, 474, 475, 476, 477, 478, 478, 479, 480, 481, 482, 483, 484, 485, 486, 487, 488, 489, 489, 490, 491, 492, 493, 494, 495, 496, 497, 498, 499, 499, 500])  
fig, ax = plt.subplots(figsize = (8,8))  
  
ax.plot(x, y, "r", linewidth=2.0)  
plt.xlabel("No. of values inserted")  
plt.ylabel("Time taken in milliseconds")  
plt.show()
```



Number of data vs insertion time

IMPLEMENTING RED BLACK TREES ON LARGE DATASET

Space Used :

- 1) RB node : key(string), rightchild, leftchild, parent, color, index(int)
- 2) Adjacency list (space : nodes + edges)
- 3) Vertices : It is an array which stores the string we input corresponding to its index.(so that access can be in constant time)

Procedure :

- 1) We are taking input and instantiating a RB node for every unique vertex in the graph.
- 2) As we insert new nodes we are assigning each node with an index value(which is initial 0 and incremented as we take input) and storing its key to an array of vertices at this assigned index.
- 3) Also simultaneously we are making an adjacency list of data.

Jure Leskovec



[SNAP for C++](#)
[SNAP for Python](#)
[SNAP Datasets](#)
[BIOSNAP Datasets](#)
[What's new](#)
[People](#)
[Papers](#)
[Projects](#)
[Citing SNAP](#)
[Links](#)
[About](#)
[Contact us](#)

 [California road network](#)

[Dataset information](#)

A road network of California. Intersections and endpoints are represented by undirected edges.

[Dataset statistics](#)

Nodes	1965206
Edges	2766607
Nodes in largest WCC	1957027 (0.996)
Edges in largest WCC	2760388 (0.998)
Nodes in largest SCC	1957027 (0.996)
Edges in largest SCC	2760388 (0.998)
Average clustering coefficient	0.0464
Number of triangles	120676
Fraction of closed triangles	0.02097
Diameter (longest shortest path)	849
90-percentile effective diameter	5e+02

IMPLEMENTING RED BLACK TREES ON LARGE DATASET

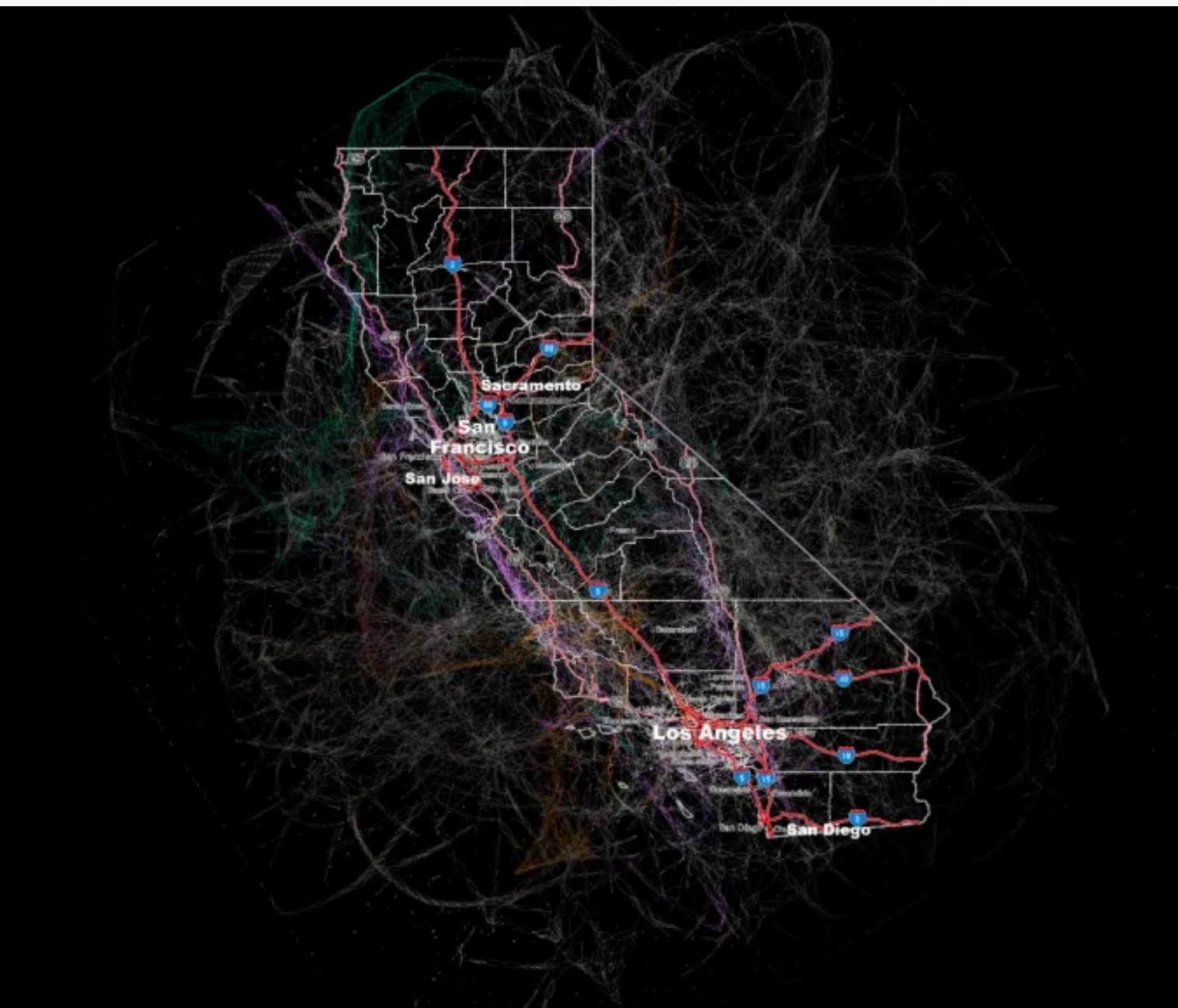
How it works :

Let's say we have a starting point ABC and ending point XYZ.

- 1) We will search ABC and XYZ in the RB tree and get its index values.
- 2) After we get 2 indexes we have an adjacency list by which we can find paths by DFS or BFS.
- 3) If we want to print a path we can get it from the vertices array as it stores string corresponding to index.

Time Taken : (averaged)

- 1) Single insert after 1900000 insertion = 0.0067 ms
- 2) Single search after 1900000 insertion = 0.0047 ms
- 3) Total time taken (insertion for all nodes) = 7.42 s



courtesy : Studentwork.prattisi.org

ADVANTAGES



Red Black Tree Code is relatively easier to understand and implement. At same time it is faster also.

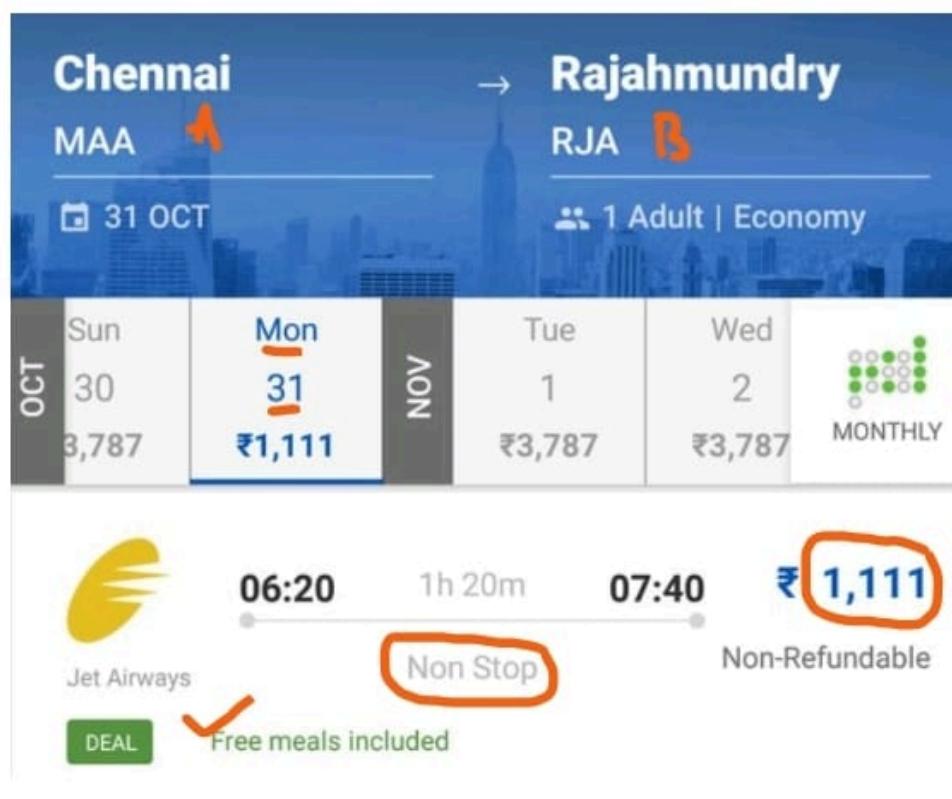


We can retrieve Information very Fast as we saw the analysis of RB Trees.



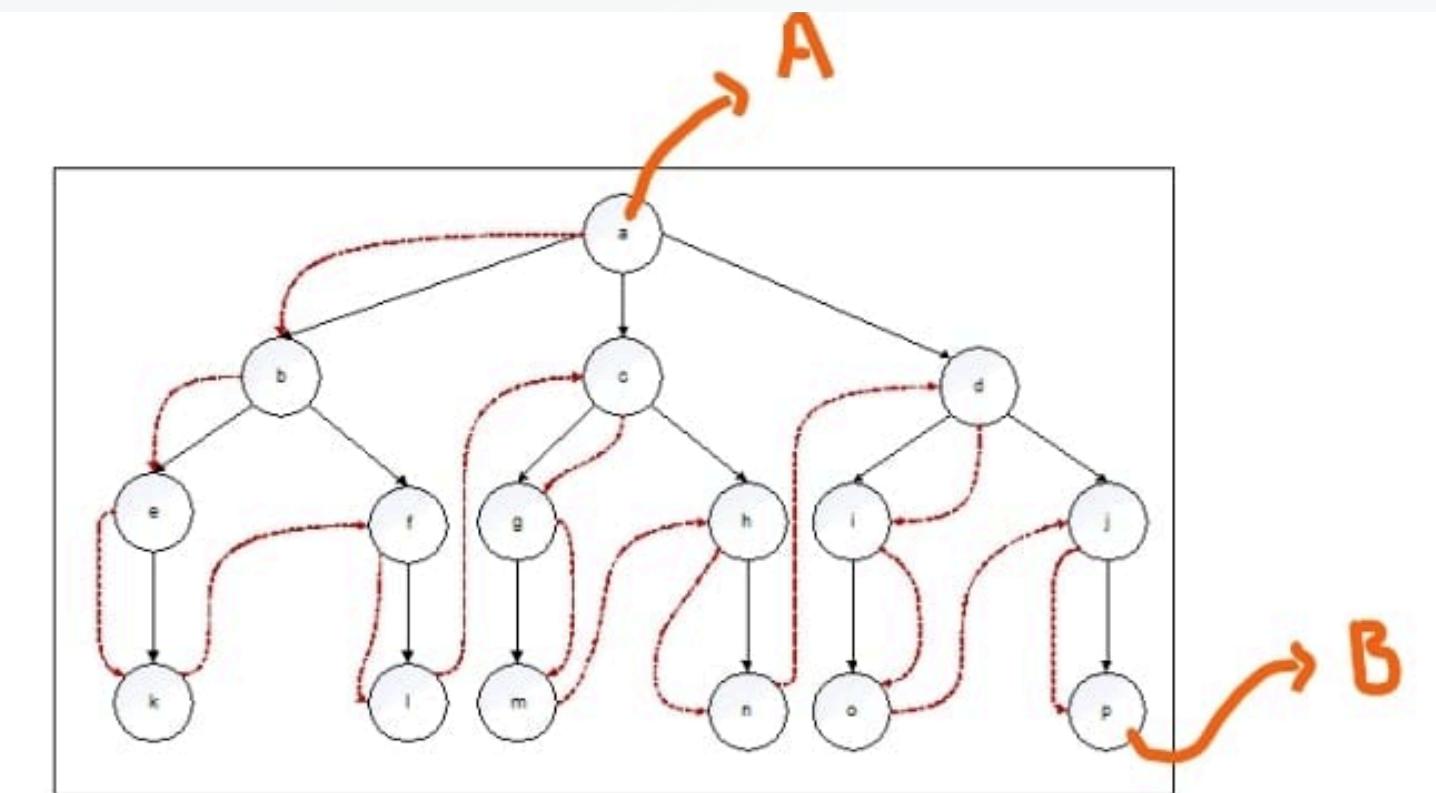
RB Trees take less steps to balance. In Max two Steps RB tree is balanced.

TRAVEL BOOKING APPS



- Searching of origin and destiny will be faster if we make a RB Tree of Stops.

- If we don't have direct transport from origin to destiny then we will try another transport link from junction which are connected to origin and this info can be stored.



SOME APPLICATIONS OF RB TREE



DATABASE
INDEXING

LIBRARY
FUNCTIONS
(EG. MAPS, SET)

LINUX KERNEL'S
COMPLETE FAIR
SCHEDULER

Github Link : https://github.com/jinangshah21/Travel-Desk-/tree/main/Travel_RBT

THANK YOU

