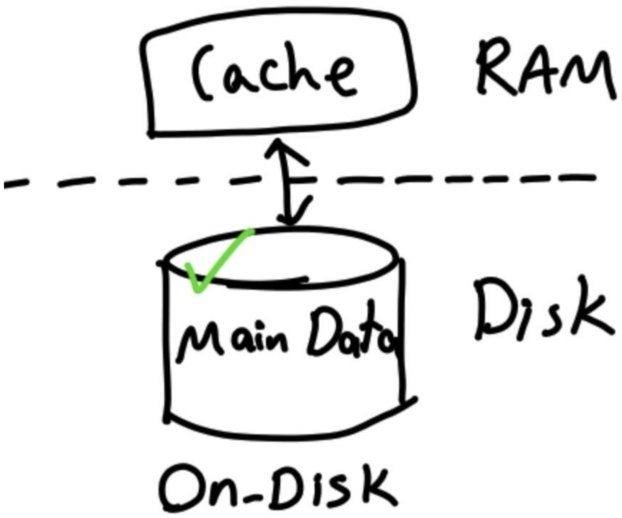
The Adaptive Radix Tree

ARTful Indexing for Main-Memory Databases

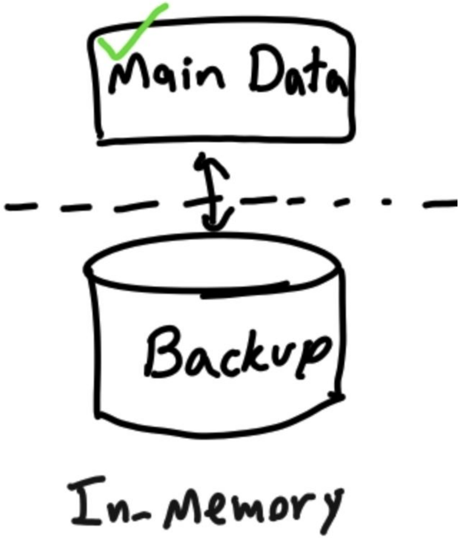
Disk-Based Database System



Bottleneck:

the overhead of disk I/O operations

Main-Memory Database System

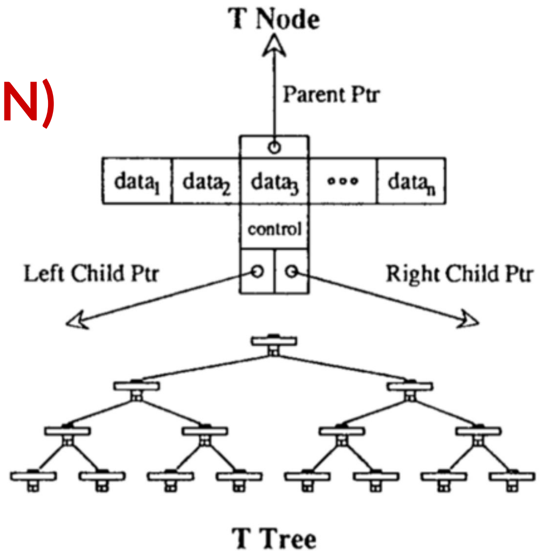


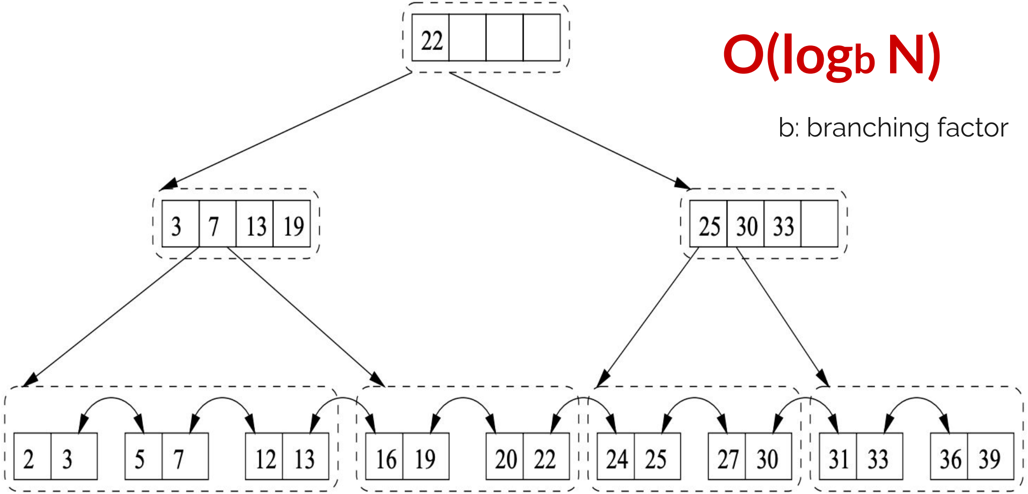
Index Efficiency

T-tree

Cache Sensitive B+ tree

$O(\log_2 N)$





$O(\log_b N)$

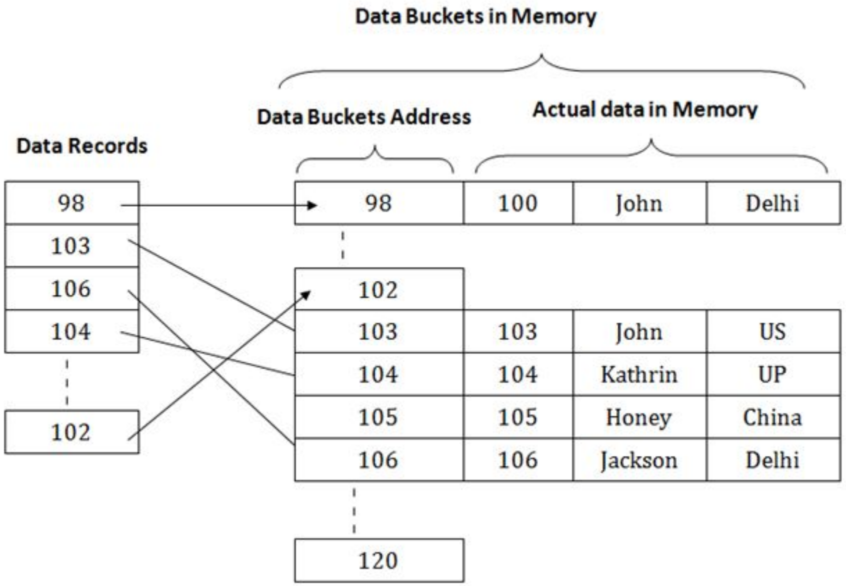
b: branching factor

Do not optimally utilize on-CPU cache

Require more expensive update operations

Is there any data structure that has better performance than O(log n)?

Only support point queries



Cannot handle growth well

Require expansive reorganization upon overflow with O(n) complexity

Hash Table !!! **O(1)**

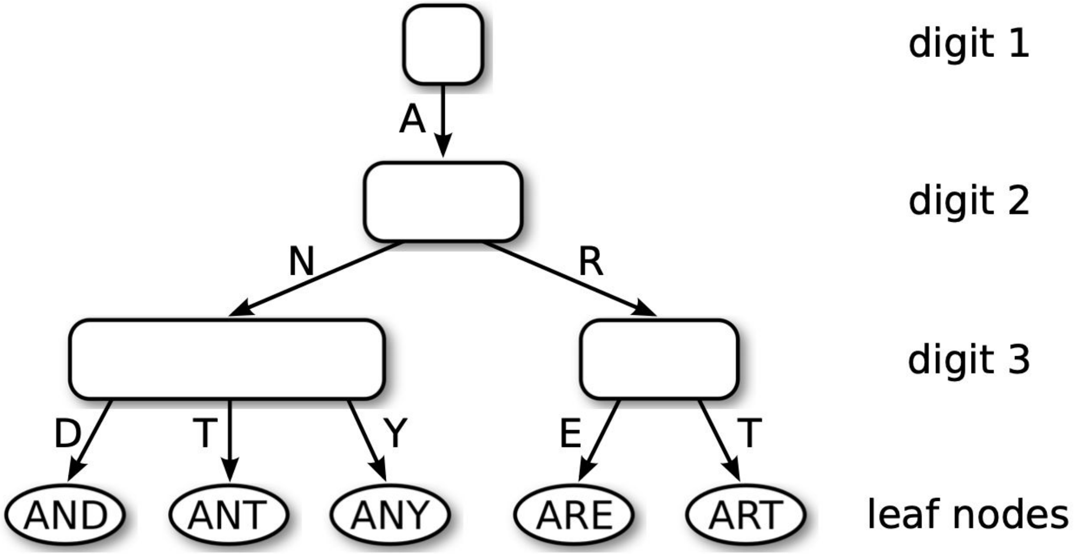
Unfortunate trade-off

Fast hash tables that only allow point queries

Fully-featured, but relatively slow, search trees.

Any other possible solution with overall better performance?

Trie / Prefix Tree / Digital Search Tree



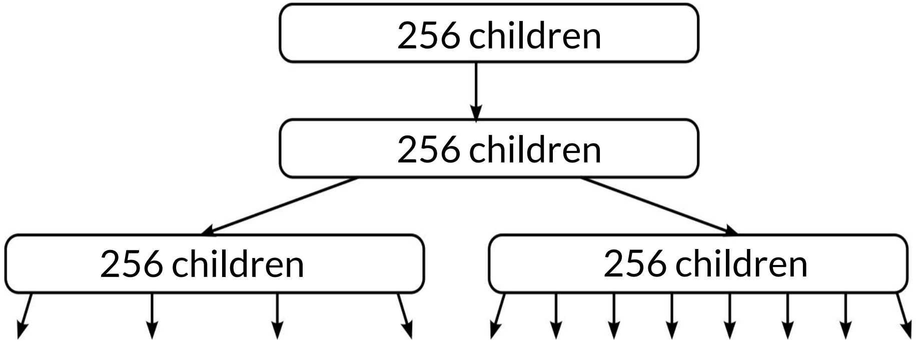
O(k)

Radix Tree

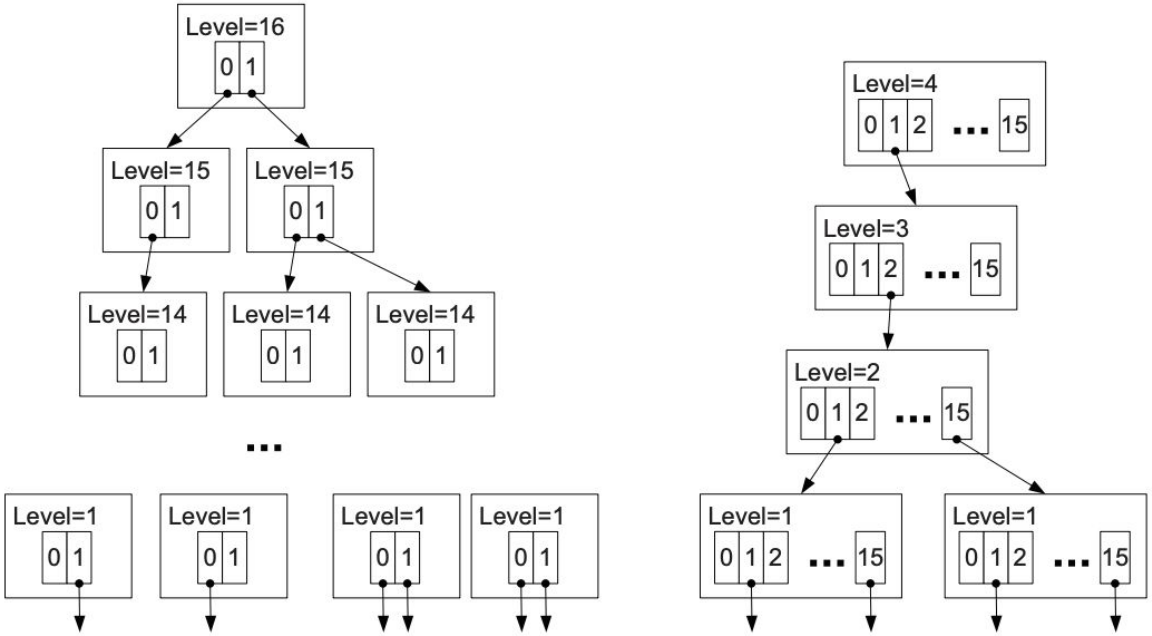
00101101: 8-bits span

00000000
00000001
00000010
...
11111110
11111111

} 2^8 children

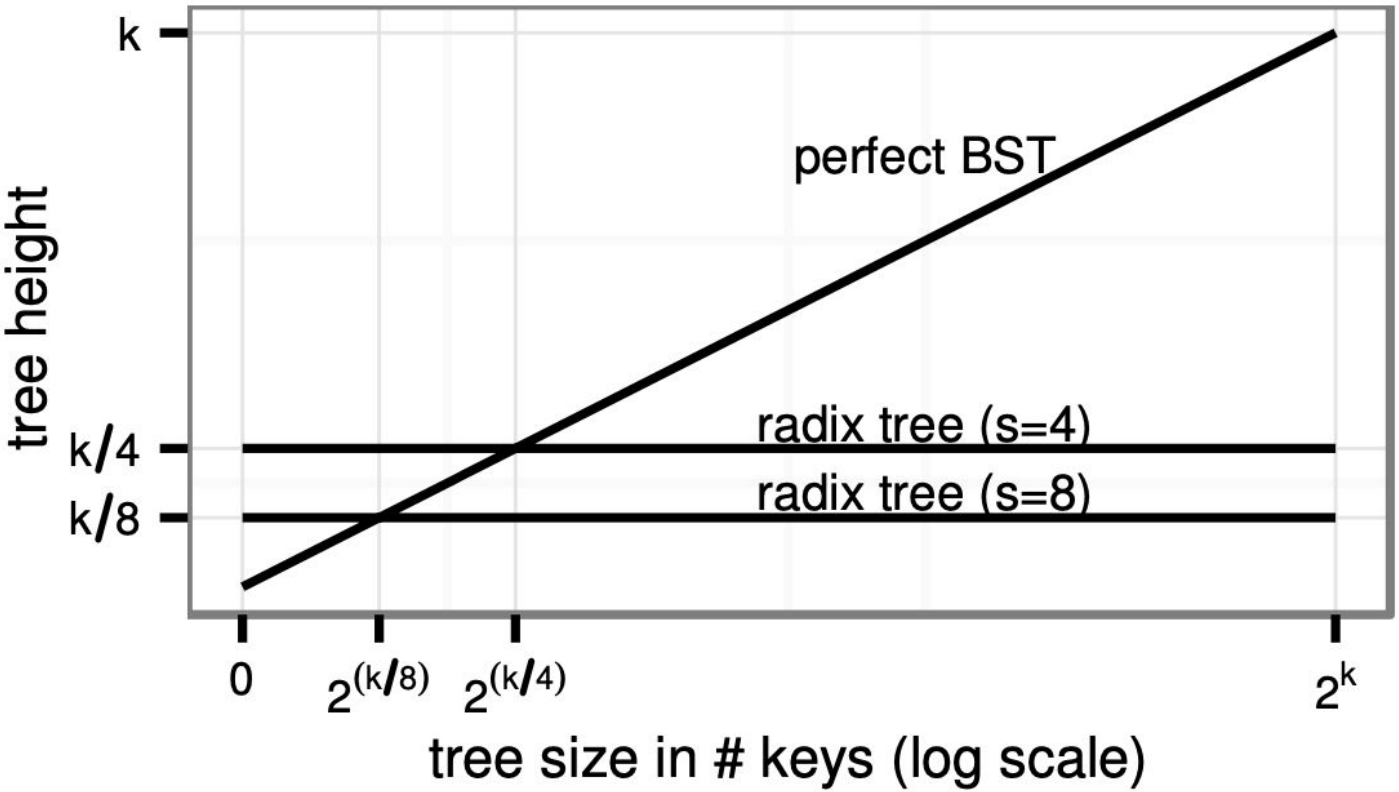


Generalized Prefix Tree



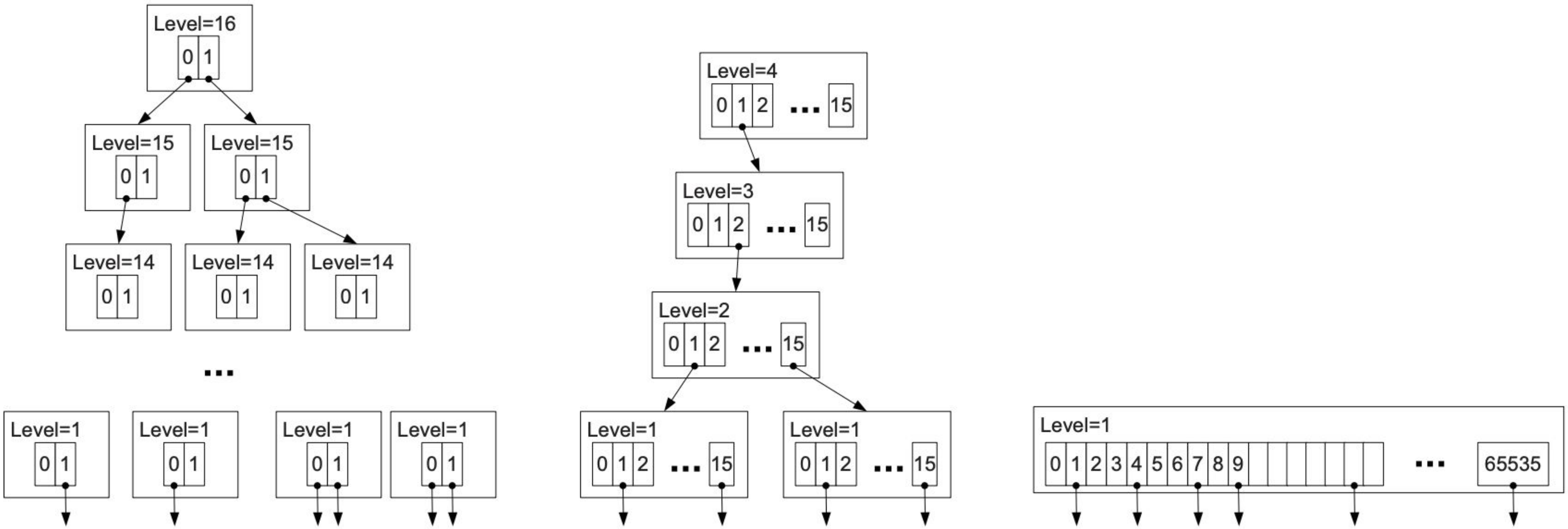
(a) $k' = 1$ ($h = 16$)

(b) $k' = 4$ ($h = 4$)



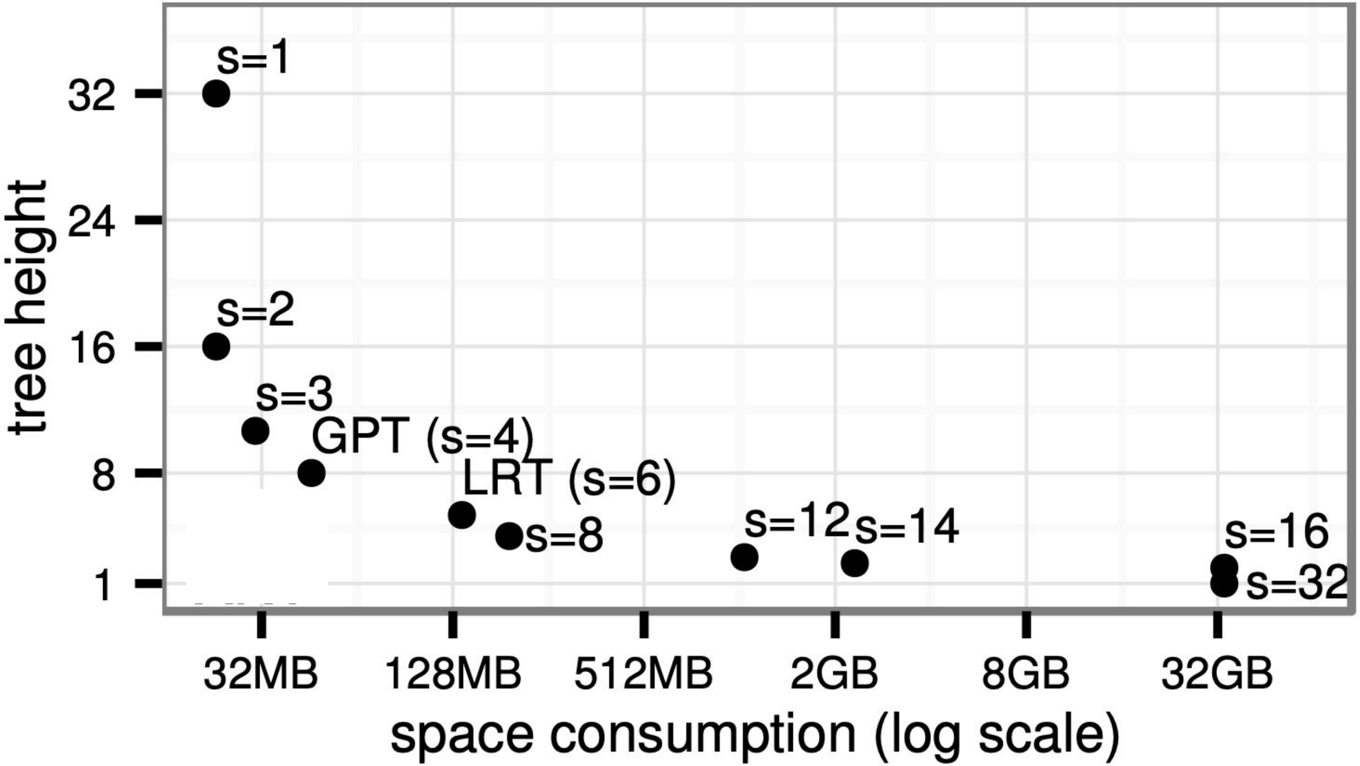
BST: Binary Search Tree   s: span parameter   k: # bits of keys

Generalized Prefix Tree



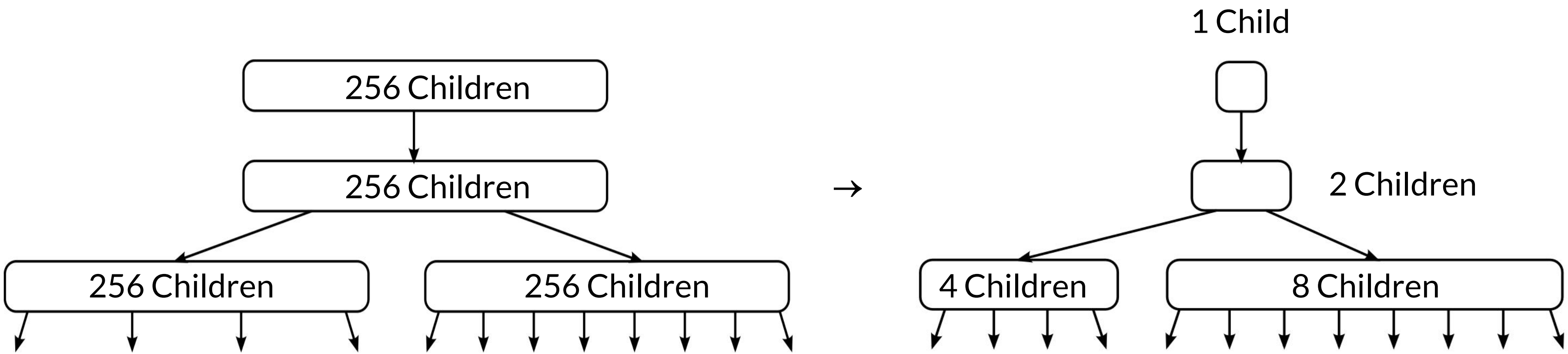
(a) $k' = 1$ $(h = 16)$ (b) $k' = 4$ $(h = 4)$ (c) $k' = 16$ $(h = 1)$

Tree Height vs. Space Consumption



GPT: Generalized Prefix Tree LRT: Linux kernel radix tree s: span parameter

Adaptive Radix Tree (ART)



Adaptive Node Design

Adaptive Radix Tree (ART)

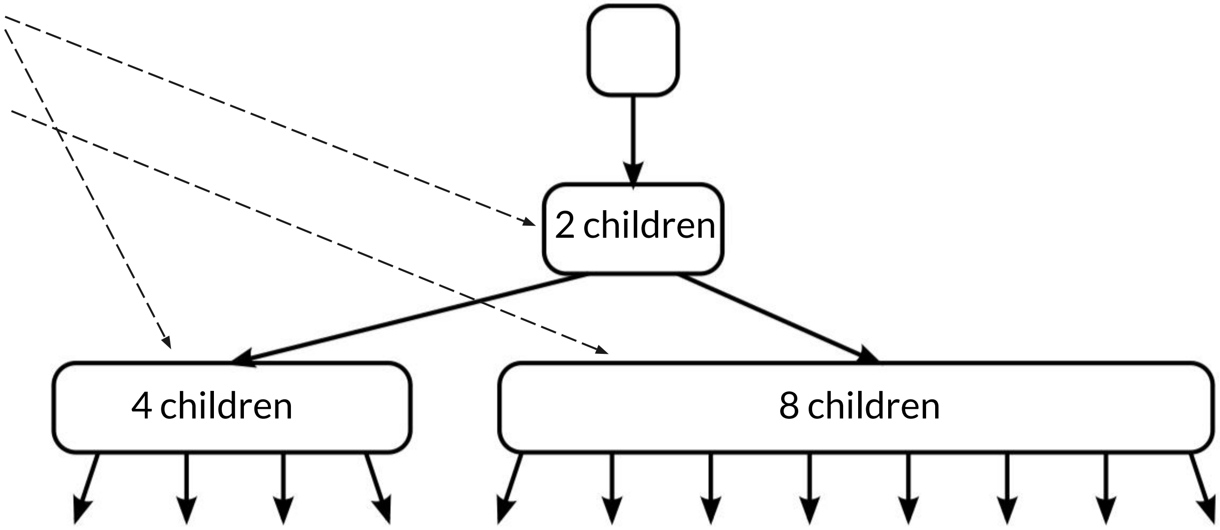
4 types of nodes:

Node4

Node16

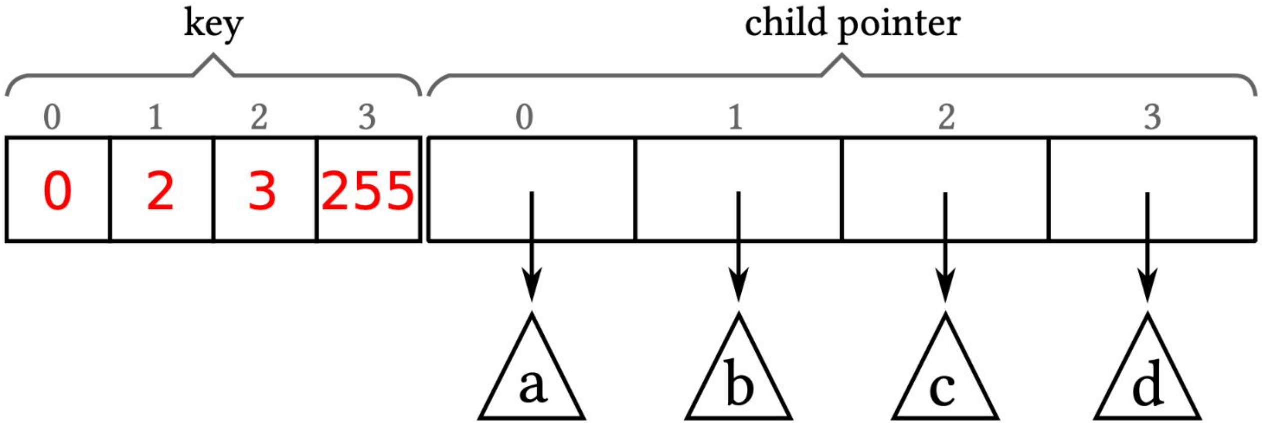
Node48

Node256

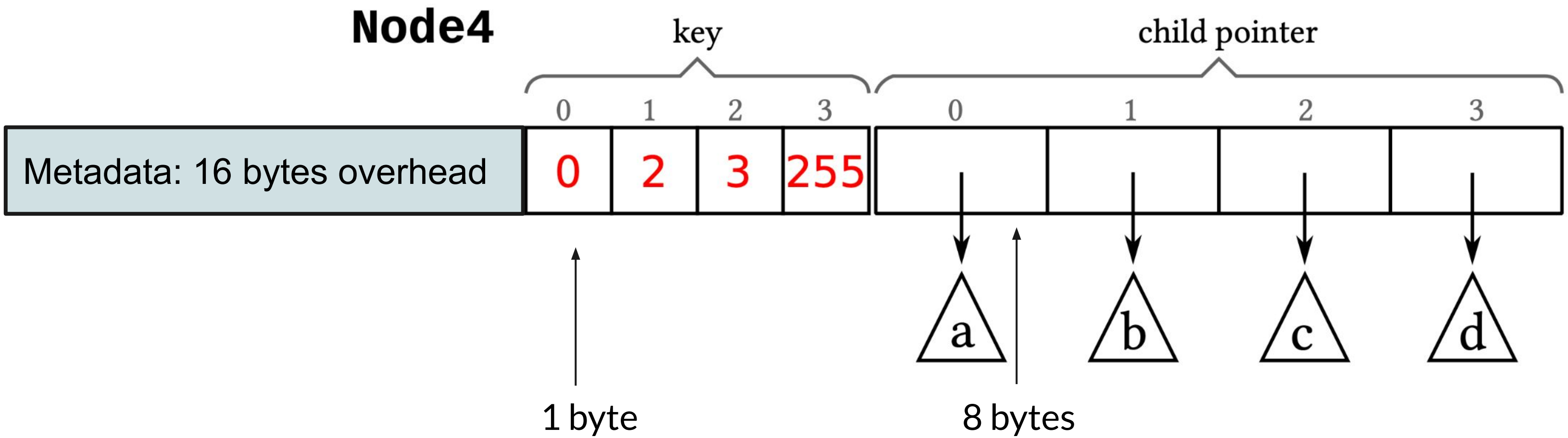


ART Inner Node Type 1

Node4



Space Consumption

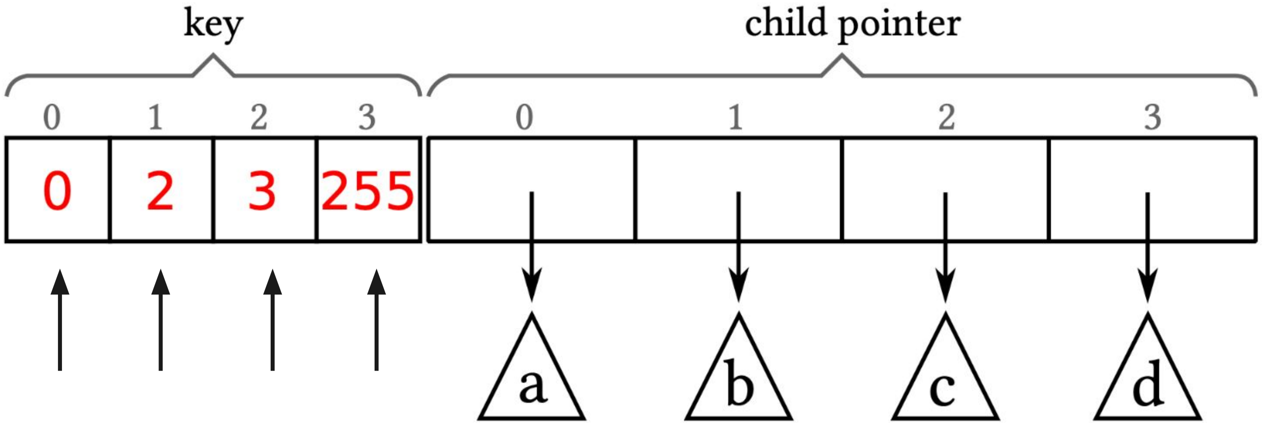


How to search the key array?

16 + 4 + 4 x 8 = 52 bytes

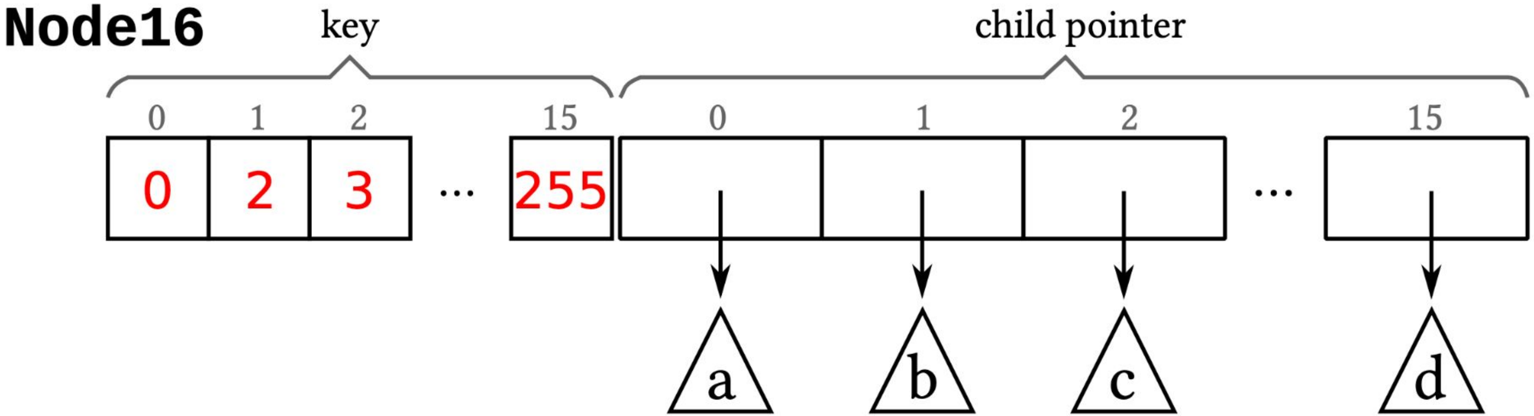
ART Inner Node Type 1

Node4



When search: Loop through

ART Inner Node Type 2



How to search the key array?

ART Inner Node Type 2

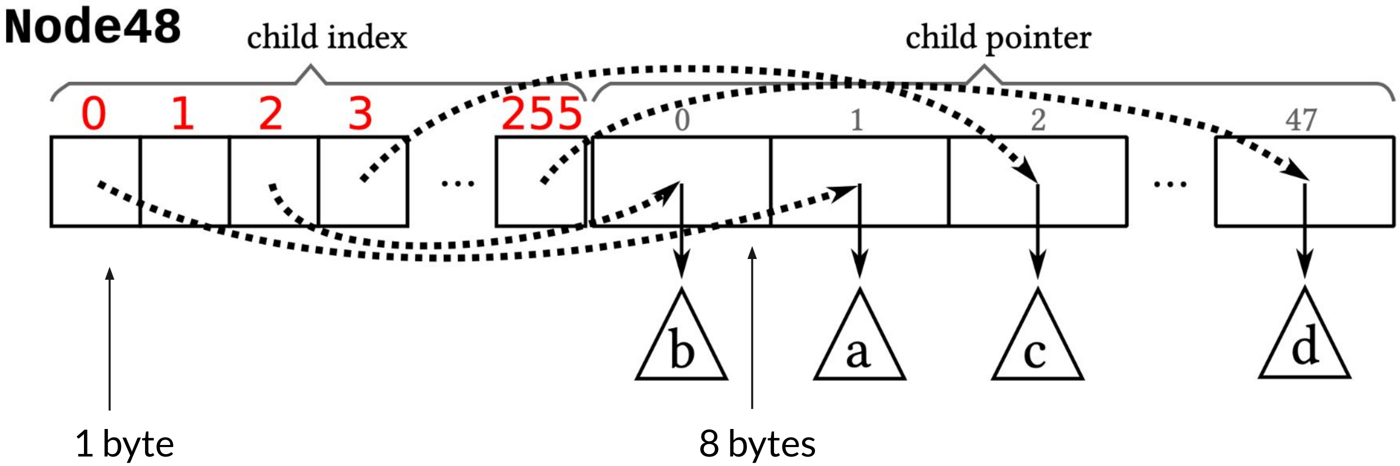
Search in 16 keys

01001100		00110101	0
01001100		00111110	0
01001100	compare	01001100	1
...		...	...
01001100		01110001	0
01001100		11001101	0

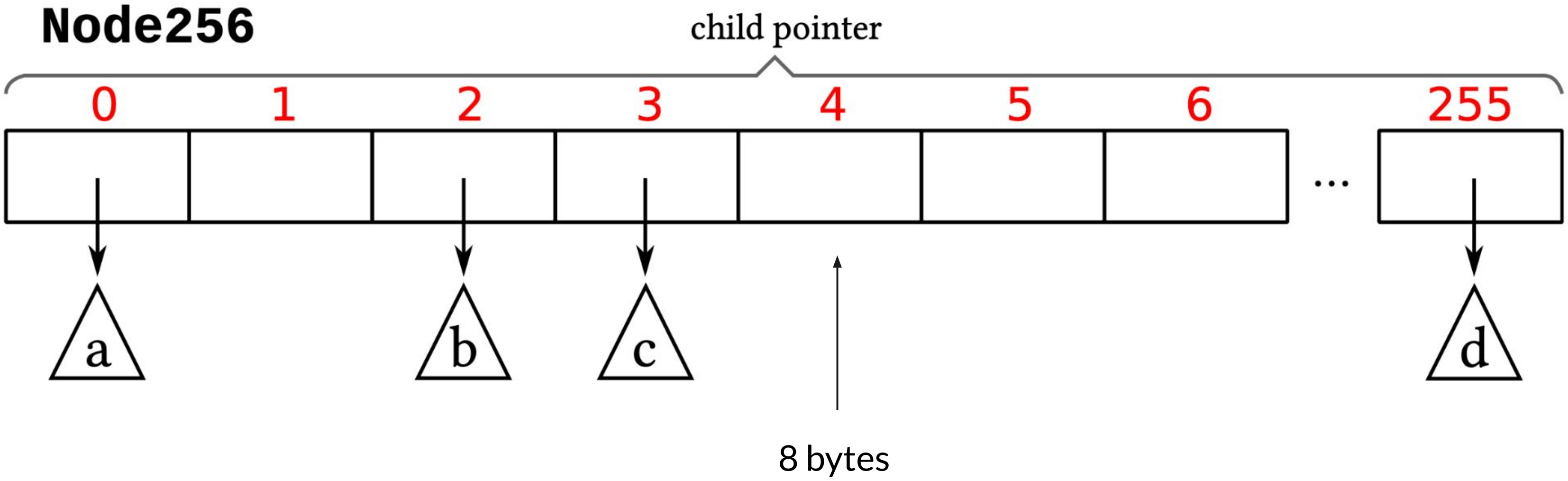
Replicate our key 16 times

Compare in parallel

ART Inner Node Type 3



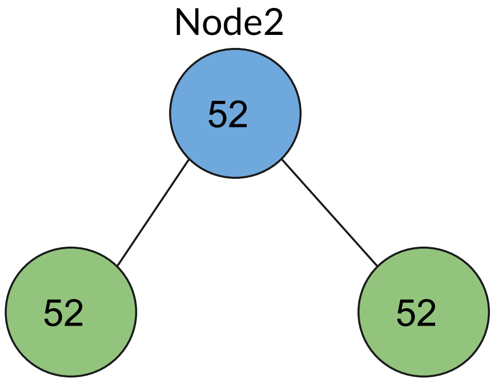
ART Inner Node Type 4



Space Consumption

Type	Children	Space (bytes)
Node4	2-4	$16 + 4 + 4 \cdot 8 = 52$
Node16	5-16	$16 + 16 + 16 \cdot 8 = 160$
Node48	17-48	$16 + 256 + 48 \cdot 8 = 656$
Node256	49-256	$16 + 256 \cdot 8 = 2064$

Space Consumption per key

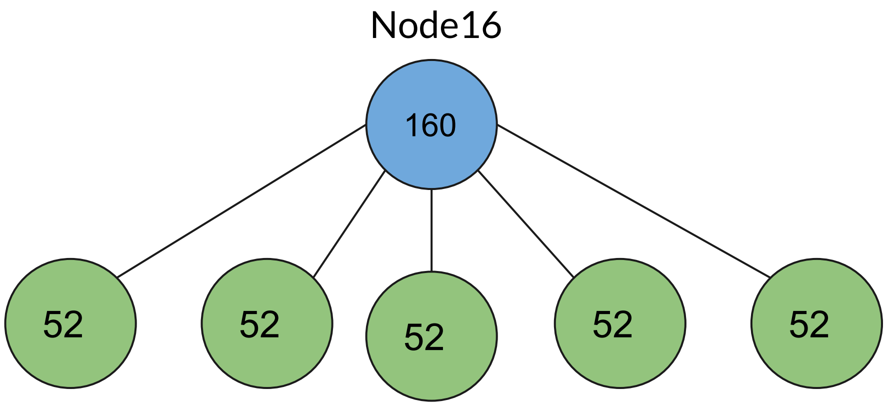


52 bytes budget/leaf

Space consumption per key is bounded by 52

Type	Minimum Children	bytes
Node4	2	52
Node16	5	160
Node48	17	656
Node256	49	2064

Space Consumption per key

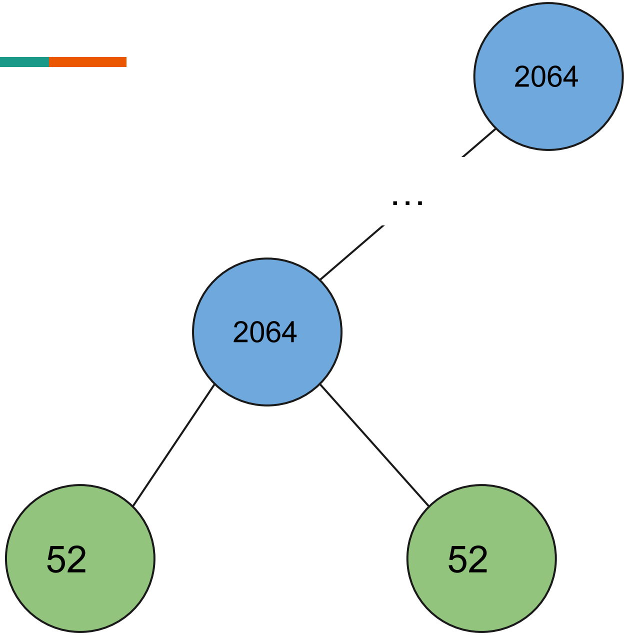


52 x 5 = 260 > 160

52 x 17 = 884 > 656
52 x 49 = 2548 > 2064

Type	Minimum Children	bytes
Node4	2	52
Node16	5	160
Node48	17	656
Node256	49	2064

Space Consumption per key



Non-adaptive radix tree:
Unbounded space consumption per key

ART Demo

Search

Insert

Delete

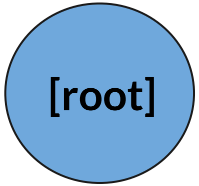
Lazy Expansion

Path Compression

ART Insertion

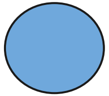
Key: smile

Value: 10

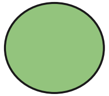


How is ART different from a trie in insertion?

Hint: Which unique technique does ART have for saving space?



Inner Node

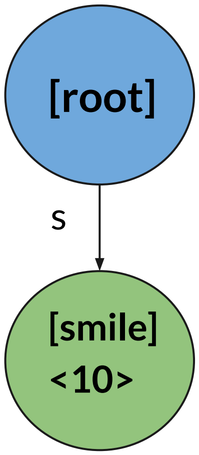


Leaf Node

ART Insertion and Path Compression

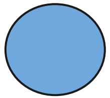
Key: smile

Value: 10

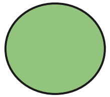


Instead of saving s->m->i->l->e sequence in individual nodes, ART path compression technique saves it in one node and reduces space consumption.

Path compression:
One inner node is removed when it has only one or no child node.



Inner Node

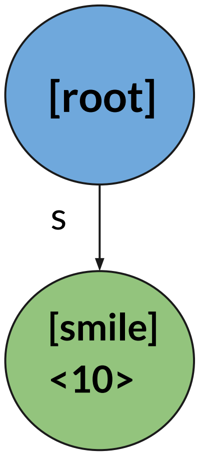


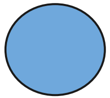
Leaf Node

ART Insertion

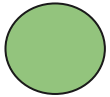
Key: smith

Value: 20





Inner Node



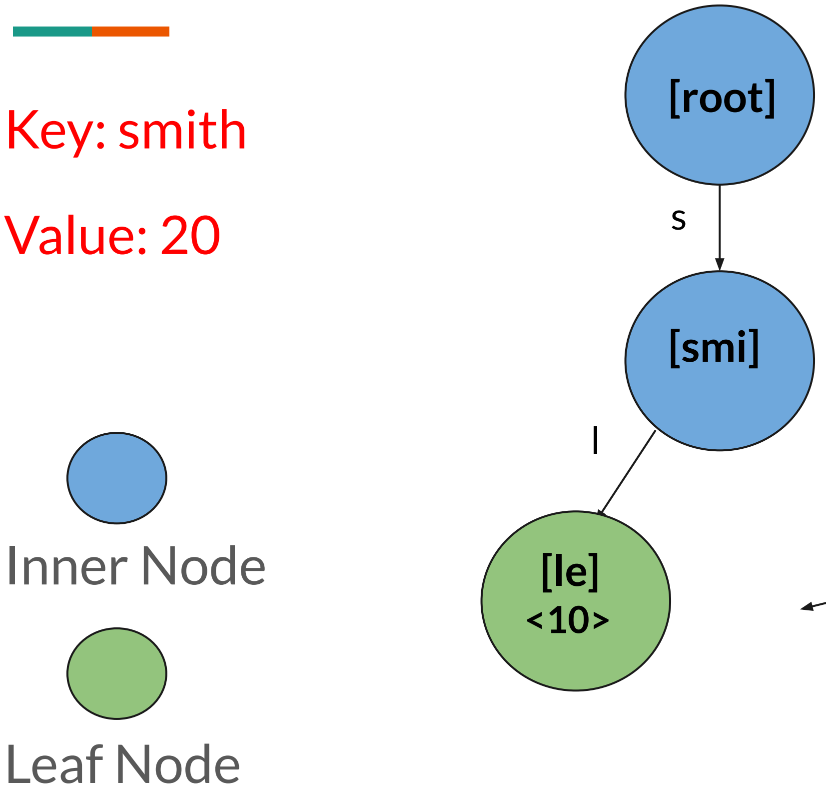
Leaf Node

What will happen next?

ART Lazy Expansion

Key: smith

Value: 20



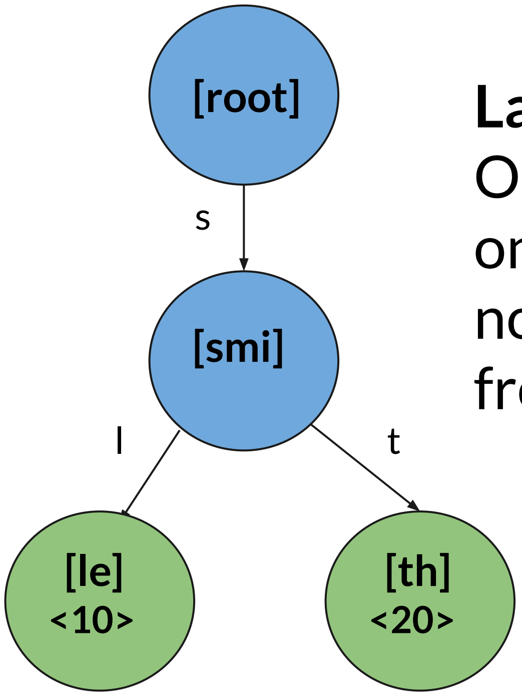
How about finding the common part of [smile] and [smith]?

What happened to the previous leaf node [smile]?

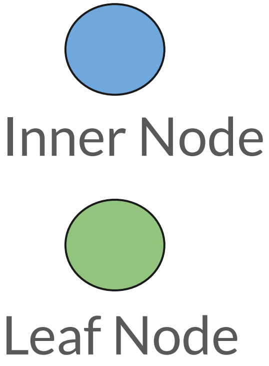
ART Lazy Expansion

Key: smith

Value: 20



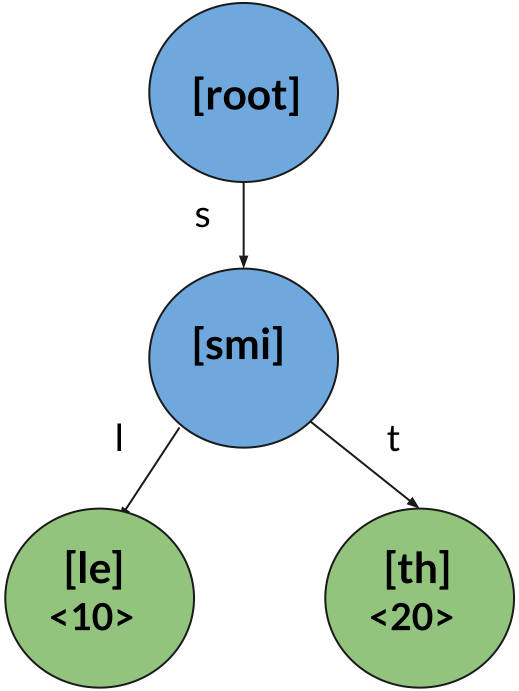
Lazy Expansion:
One new inner node is created only when at least two leaf nodes need it to distinguish from.

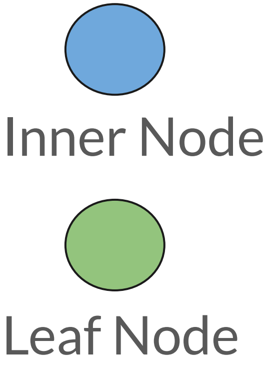


ART Insertion

Key: smiley

Value: 30



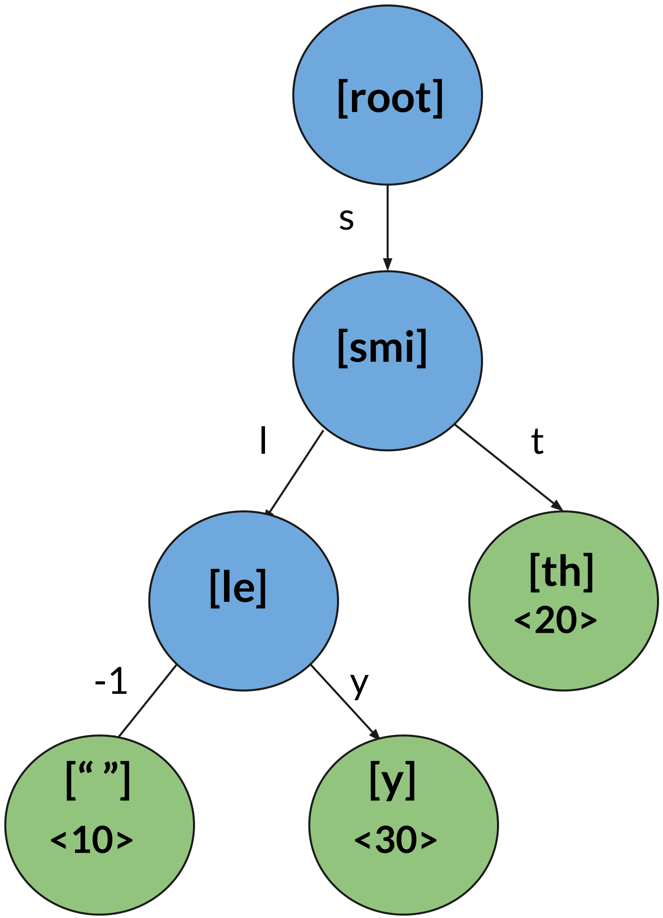


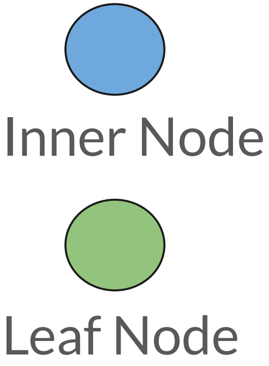
What's next if I want to insert a new key smiley?

ART Lazy Expansion

Key: smiley

Value: 30



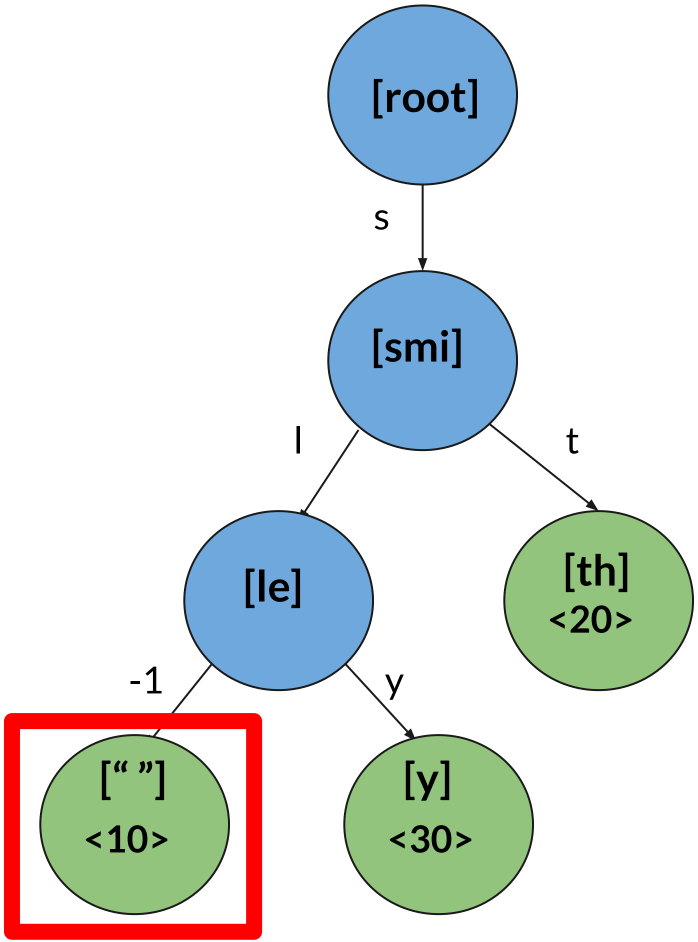


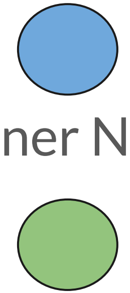
Another lazy expansion!

What’s special this time?

ART Deletion

Key: smile





Inner Node

Leaf Node

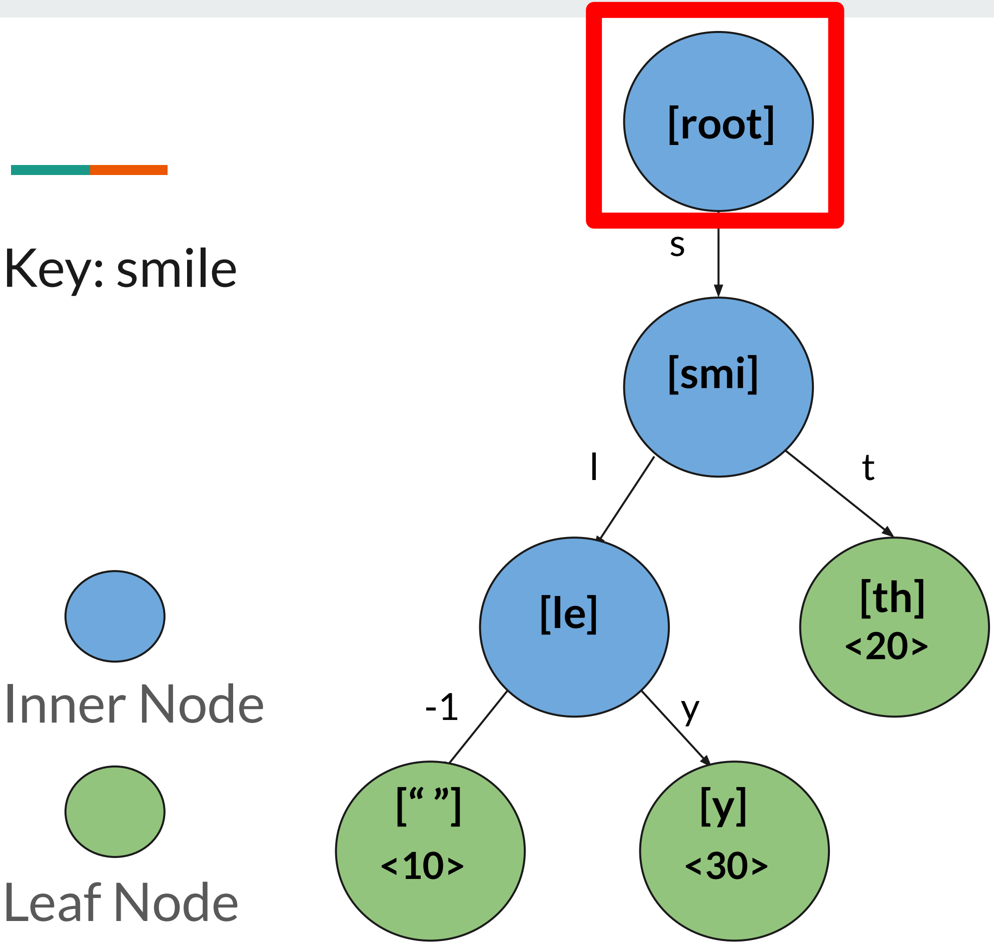
Edge case: An empty key.

If I delete it, what should happen?

What’s the first step before the deletion?

ART Search

Key: smile



Is this node null?

Is this node a leaf node?

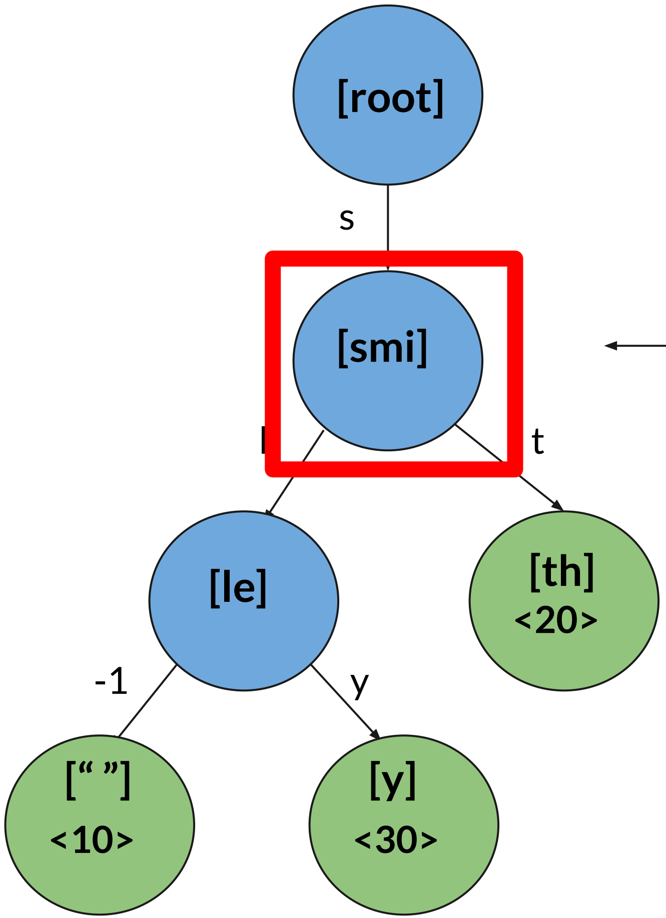
Is this node a inner node?

Searching from the root...

ART Search

Key: smile

Recursively searching...



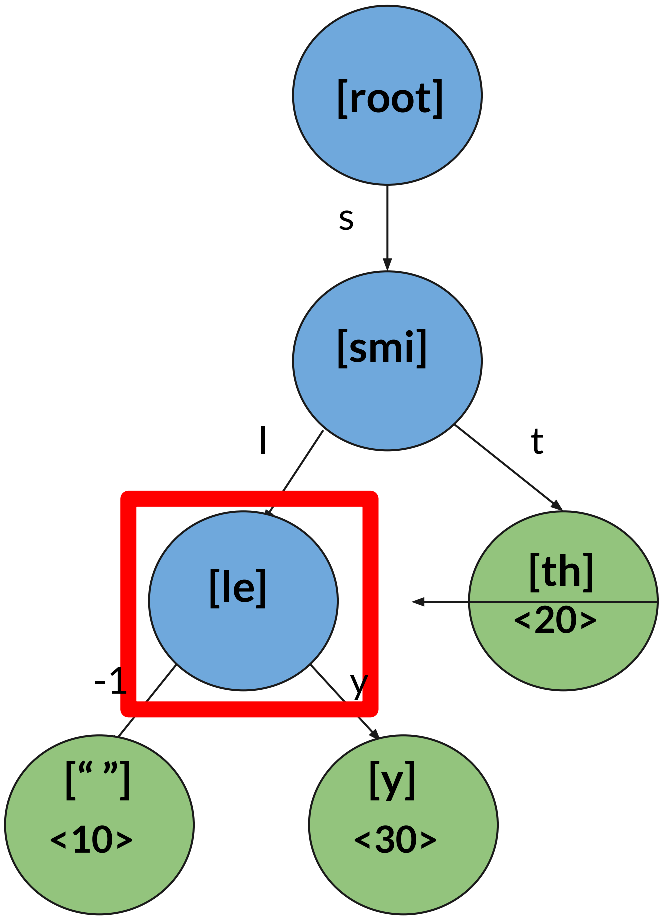
Match?

Inner Node

Leaf Node

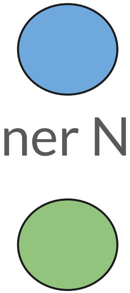
ART Search

Key: smile



Recursively searching...

Match?



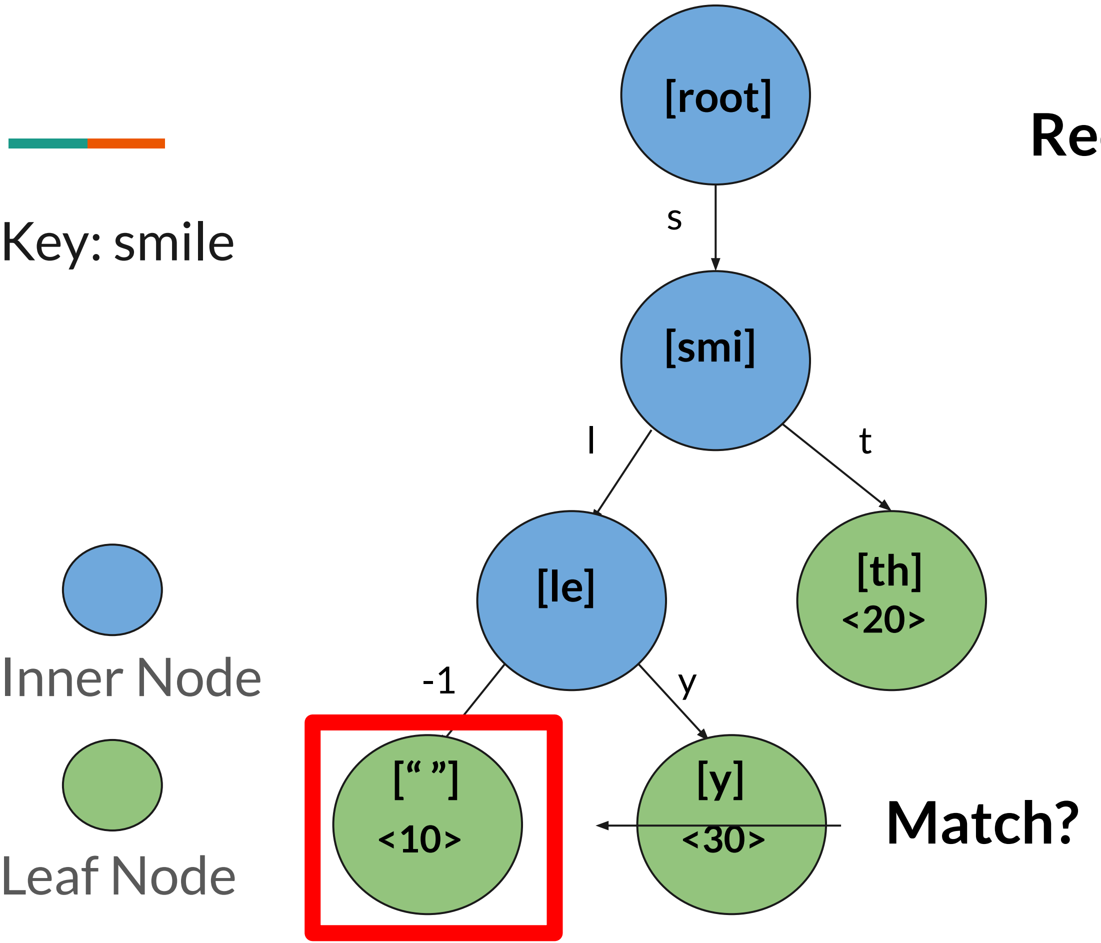
Inner Node

Leaf Node

ART Search

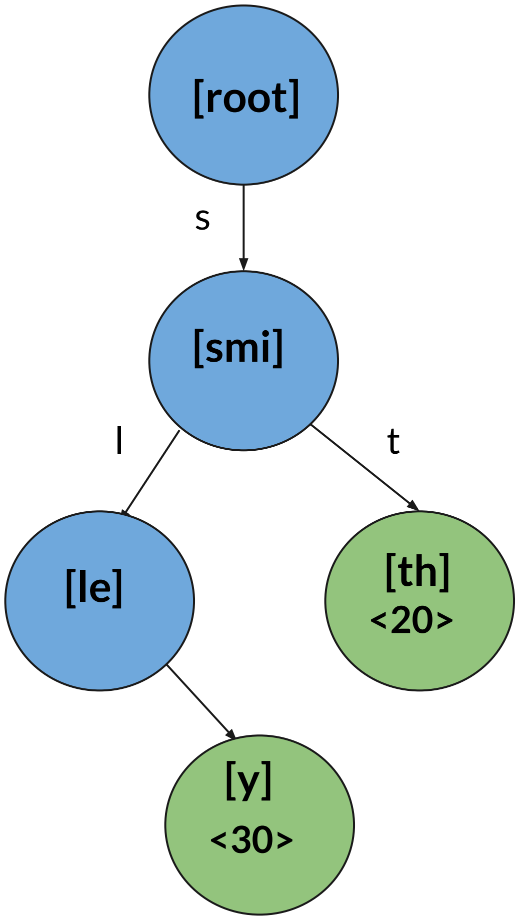
Recursively searching...

Key: smile



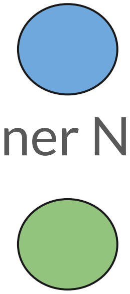
ART Deletion

Key: smile



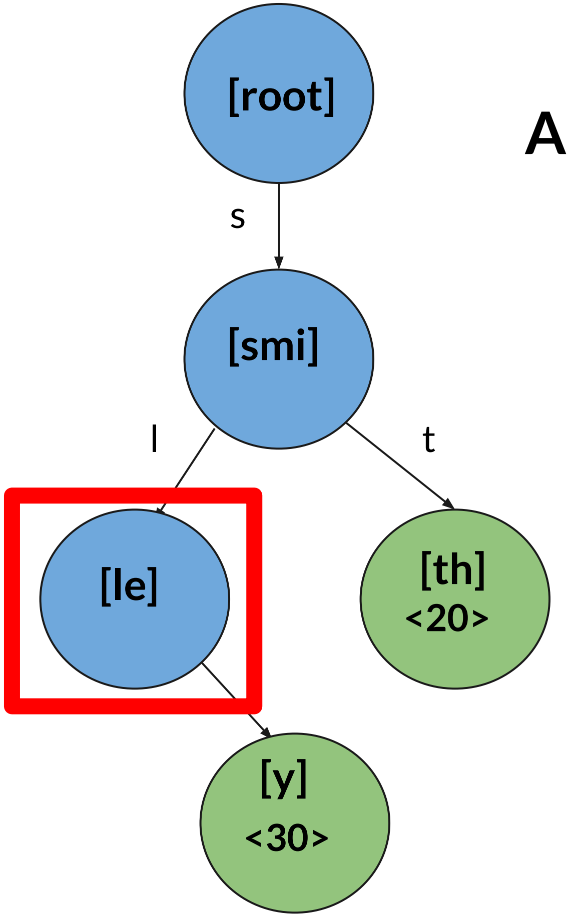
The inner node [le] now only has one child node.

What's next?



ART Path Compression

Key: smile



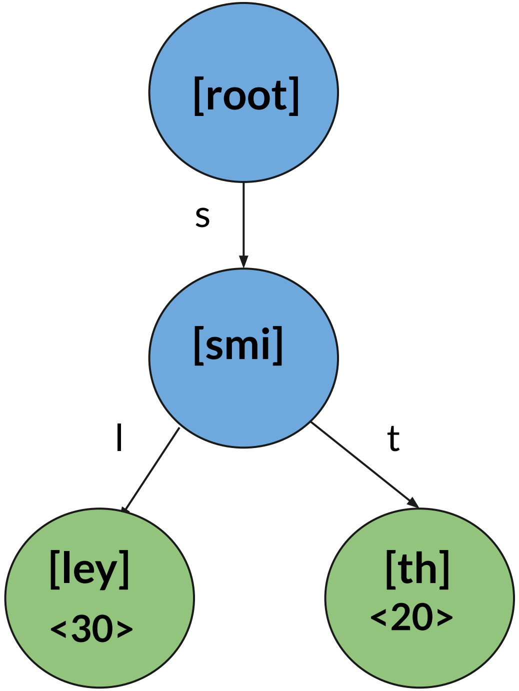
Another path compression!

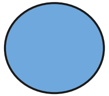
Inner Node

Leaf Node

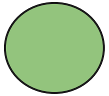
ART Path Compression

Key: smile





Inner Node



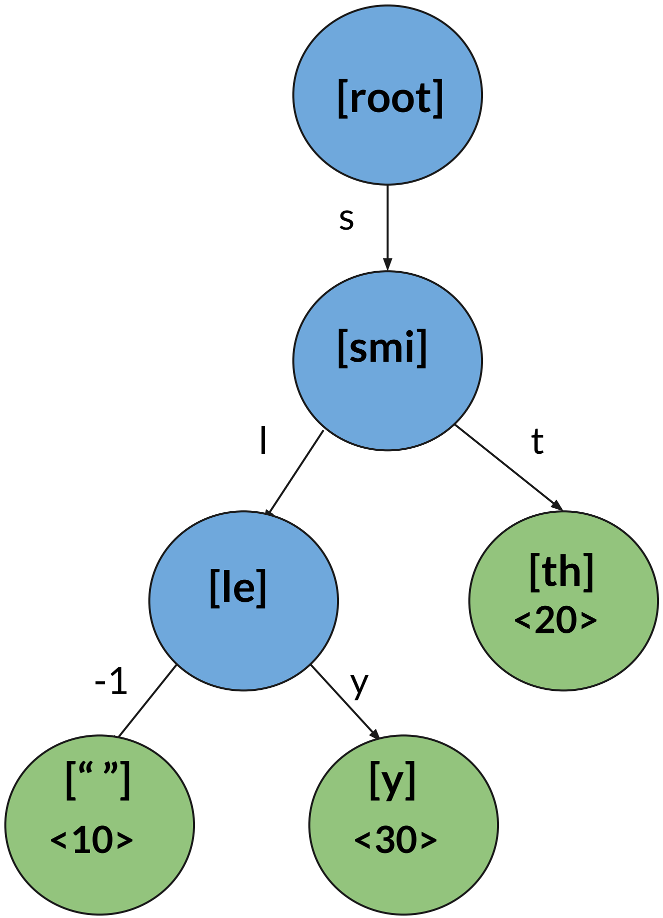
Leaf Node

ART Bulk Loading

Key 1: smile

Key 2: smith

Key 3: smiley



Bulk load all nodes.

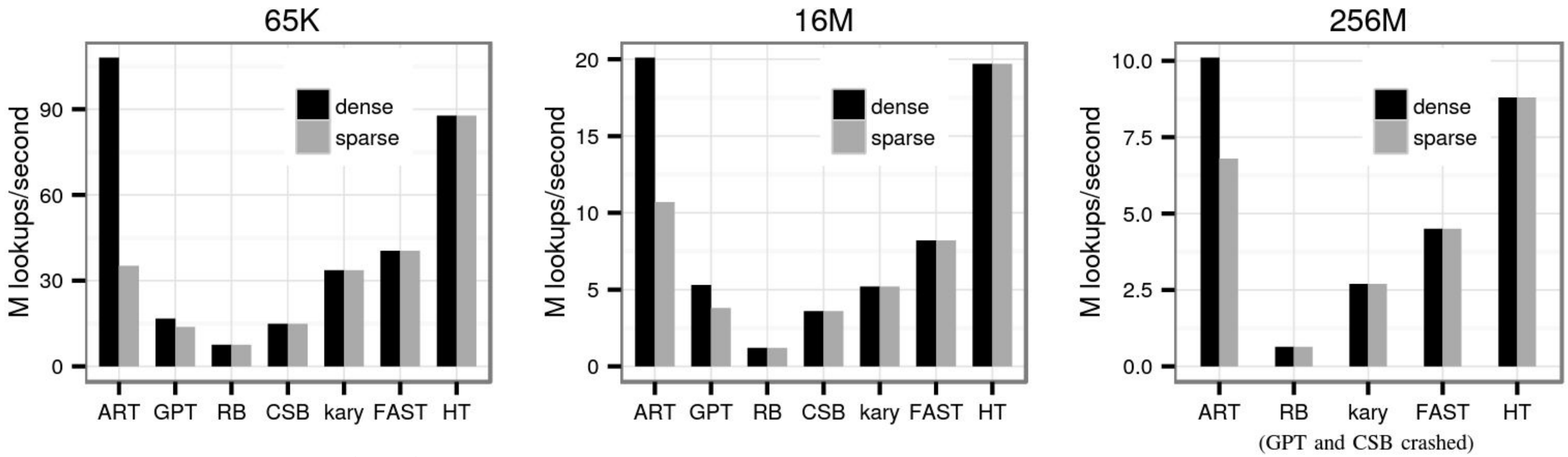
Starts from the first byte of each key and go through all keys to find the most appropriate inner node type.

Build ART while perform radix sort -> Higher time complexity.

Evaluation - Random Search

Benchmark against

GPT: Generalized Prefix Tree	RB: red-black tree
CSB: Cache-Sensitive B+-tree	HT: hash table
kary: k-ary search tree	FAST: Fast Architecture Sensitive Tree



Dense keys: Range(1, n)

k1	k2	k3	k4	k5

Sparse keys: Each bit be randomly 0 or 1

k1		k3		k5

Evaluation - Mispredictions

HT: Hash Table

FAST: Fast Architecture Sensitive Tree

TABLE III
PERFORMANCE COUNTERS PER LOOKUP.

	65K			16M		
	ART (d./s.)	FAST	HT	ART (d./s.)	FAST	HT
Cycles	40/105	94	44	188/352	461	191
Instructions	85/127	75	26	88/99	110	26
Misp. Branches	0.0/0.85	0.0	0.26	0.0/0.84	0.0	0.25
L3 Hits	0.65/1.9	4.7	2.2	2.6/3.0	2.5	2.1
L3 Misses	0.0/0.0	0.0	0.0	1.2/2.6	2.4	2.4

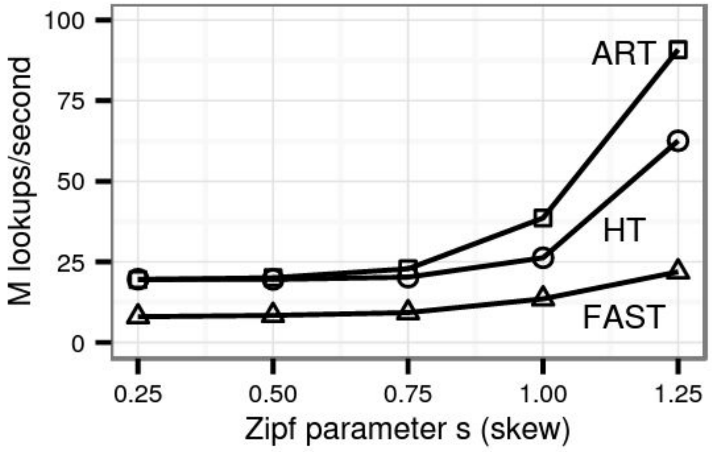
0 miss prediction branches for ART dense keys!
0.5x cache miss rate for dense keys

Evaluation - Caching Effects

HT: Hash Table

FAST: Fast Architecture Sensitive Tree

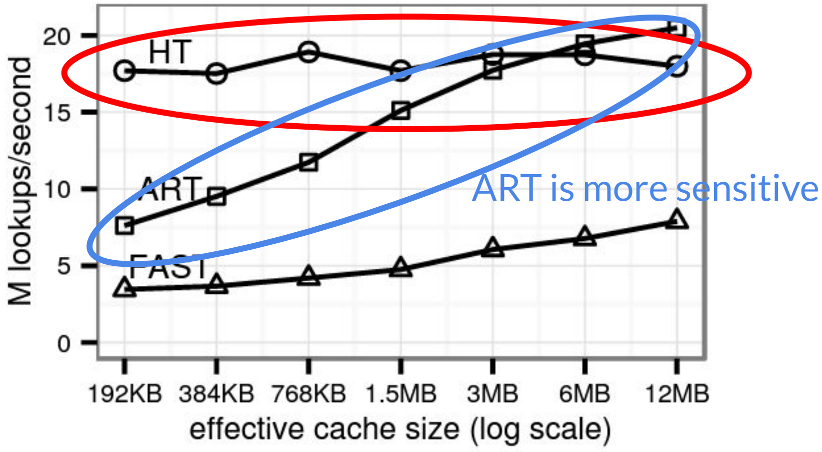
To add temporal locality,
Access skewed data to utilize the processor cache



Hit% increases

Competing Memory Access

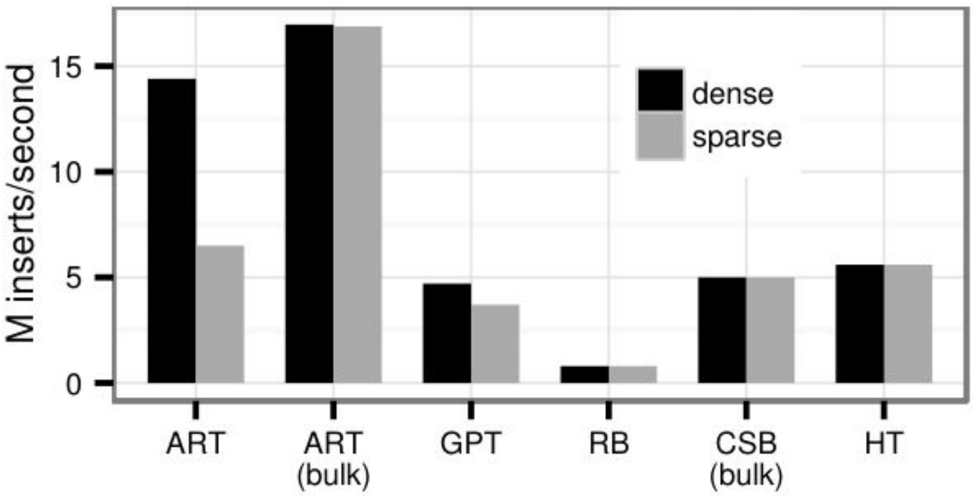
HT not affected!



Evaluation - Update

GPT: Generalized Prefix Tree | RB: red-black tree
CSB: Cache-Sensitive B+-tree | HT: hash table

Insertion Performance For 16M keys



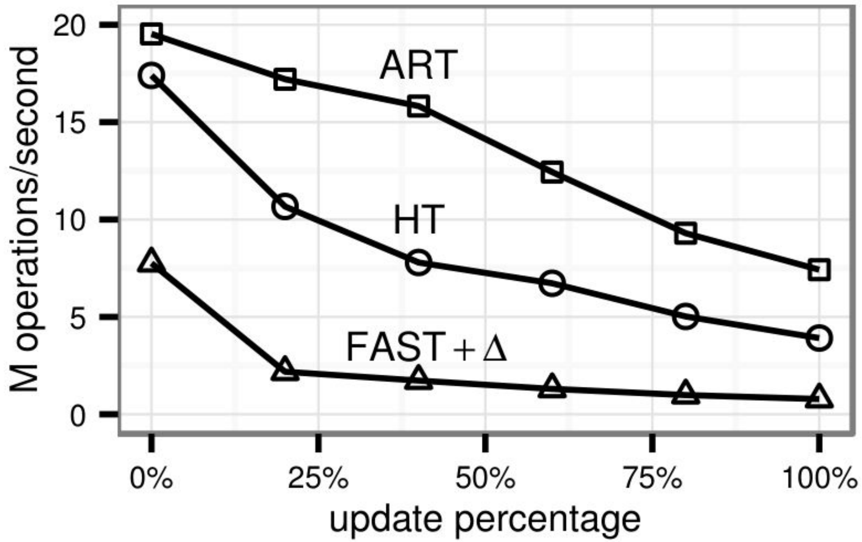
Bulk insertion: Sparse keys -> Dense keys
2.5x faster!

Evaluation - Search & Update

HT: Hash Table

FAST: Fast Architecture Sensitive Tree

Random Insertion, Deletion, Lookups



Δ is RB tree to help FAST store differences and merge later

Evaluation - End to End

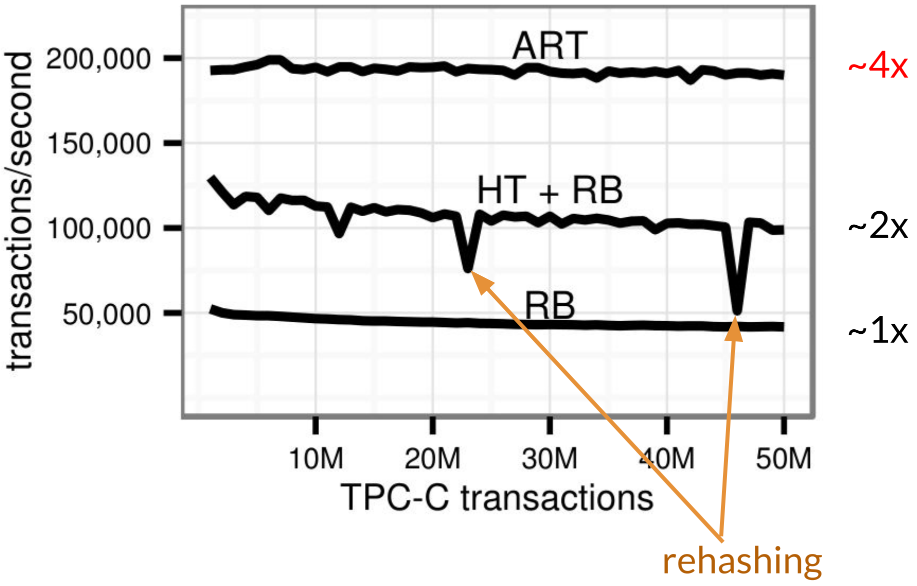
RB: red-black tree HT: hash table

TPC-C OLTP Workload

Merchandising company

Orders, Customers, Stocks

Manage, Sell, Distribute products

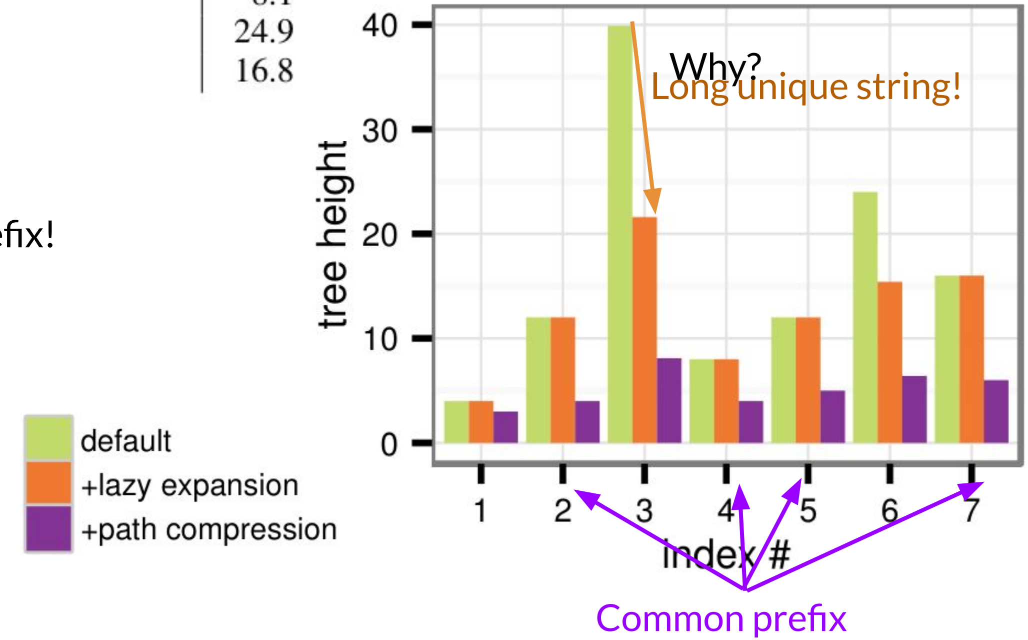


https://www.hammerdb.com/docs3.3/ch03s05.html

Evaluation - Space Consumption

#	Relation	Cardinality	Attribute Types	Space
1	item	100,000	int	8.1
2	customer	150,000	int,int,int	8.3
3	customer	150,000	int,int,varchar(16),varchar(16),TID	32.6
4	stock	500,000	int,int	8.1
5	order	22,177,650	int,int,int	8.1
6	order	22,177,650	int,int,int,int,TID	24.9
7	orderline	221,712,415	int,int,int,int	16.8

Path compression is good for common prefix!
Both are good for long strings.



Discussion

Thank You!

Q&A