BOSTON UNIVERSITY

CS 561: Data Systems Architectures

class 13

Adaptive Radix Trees

Prof. Manos Athanassoulis

https://bu-disc.github.io/CS561/

Do we have a quiz today … ?

Yes!

At the end!

BOSTON UNIVERSITY

Student Discussion Reminder

Presenters will be giving a full presentation of the main paper

Critics and **Proponents** will participate in a live Q&A during presentation (criticizing and supporting the paper) + everyone should chime in!

Critics will also present a few slides with the core critique/feedback

Proponents will address this feedback

Everyone is expected to participate in the discussion!

BOSTON UNIVERSITY

Indexing is key to database performance

B$^+$ Trees and **LSM-Trees** dominate disk-based indexes

Hash indexes and optimized search trees are common for in-memory

BUT

Hash indexes are unordered (no range queries)

Search trees are slower than Hash indexes for point queries

can we build a better compromise?

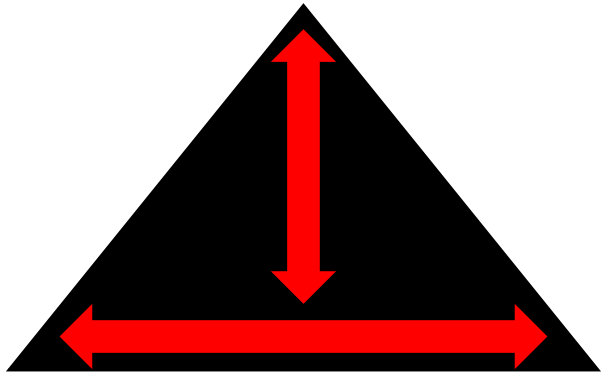
BOSTON UNIVERSITY

Increasing data size

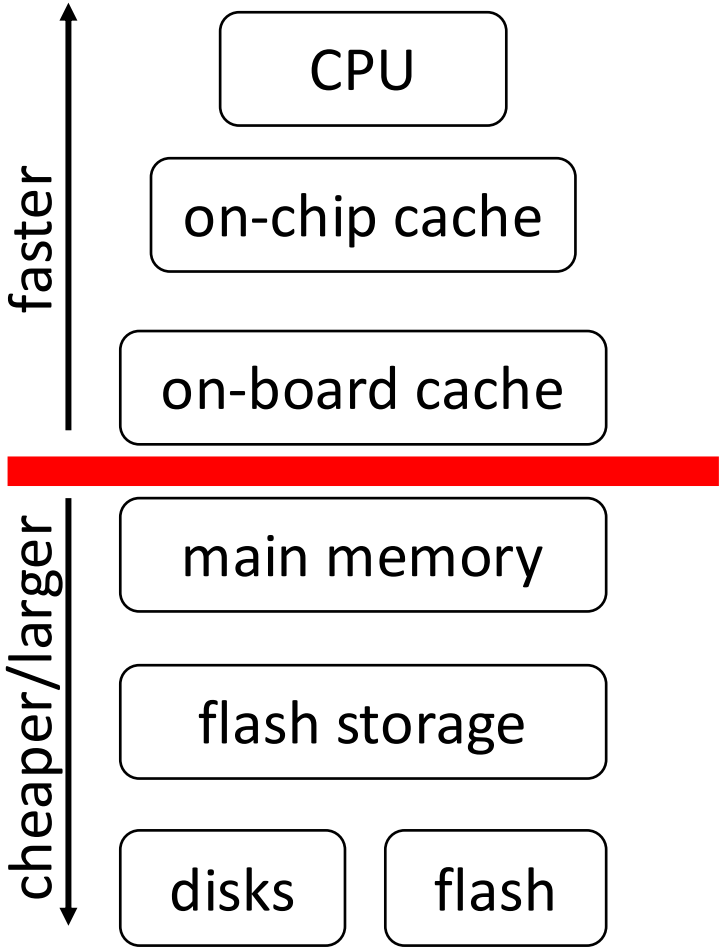
Search trees size (tree height and width) grows with data size!

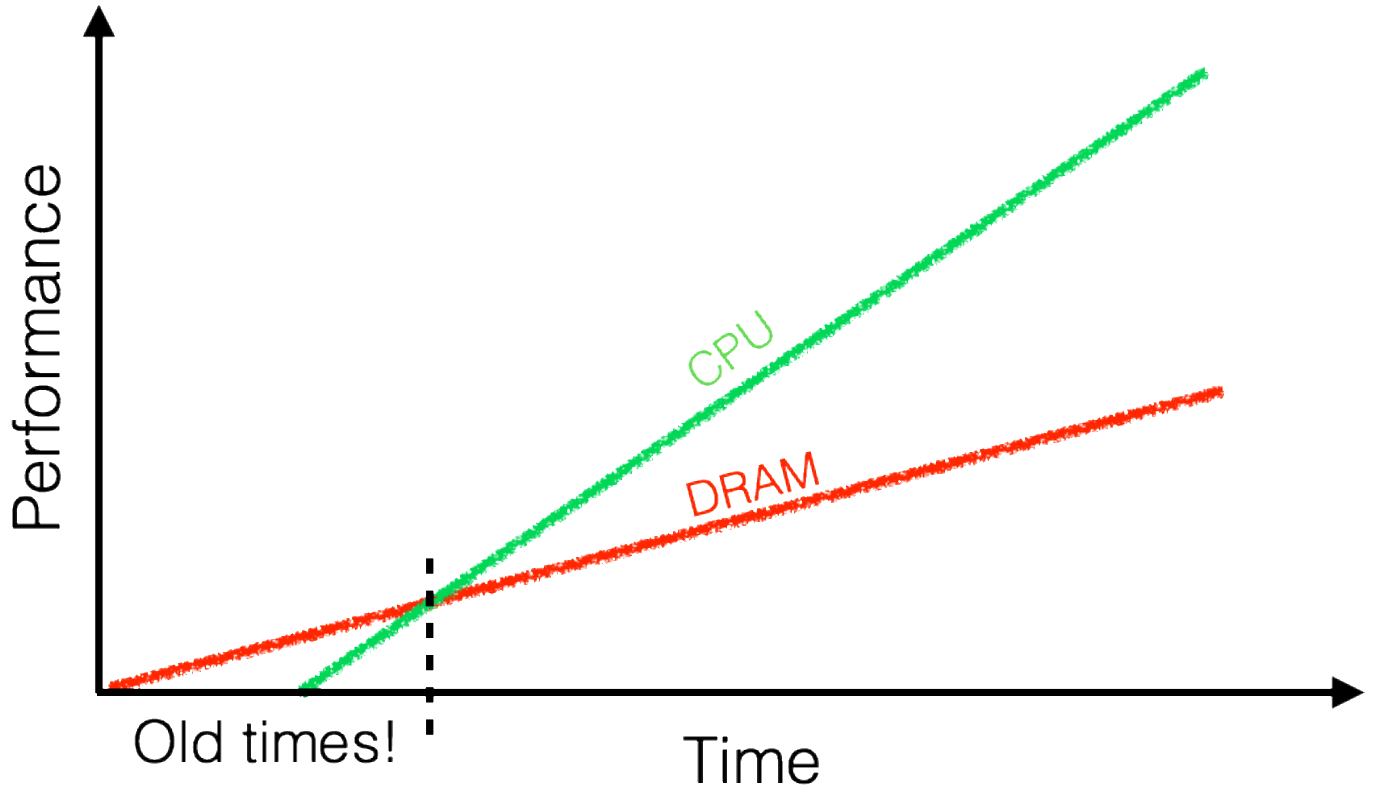
So, it quickly **does not fit** in cache or in memory

Why is that problem?

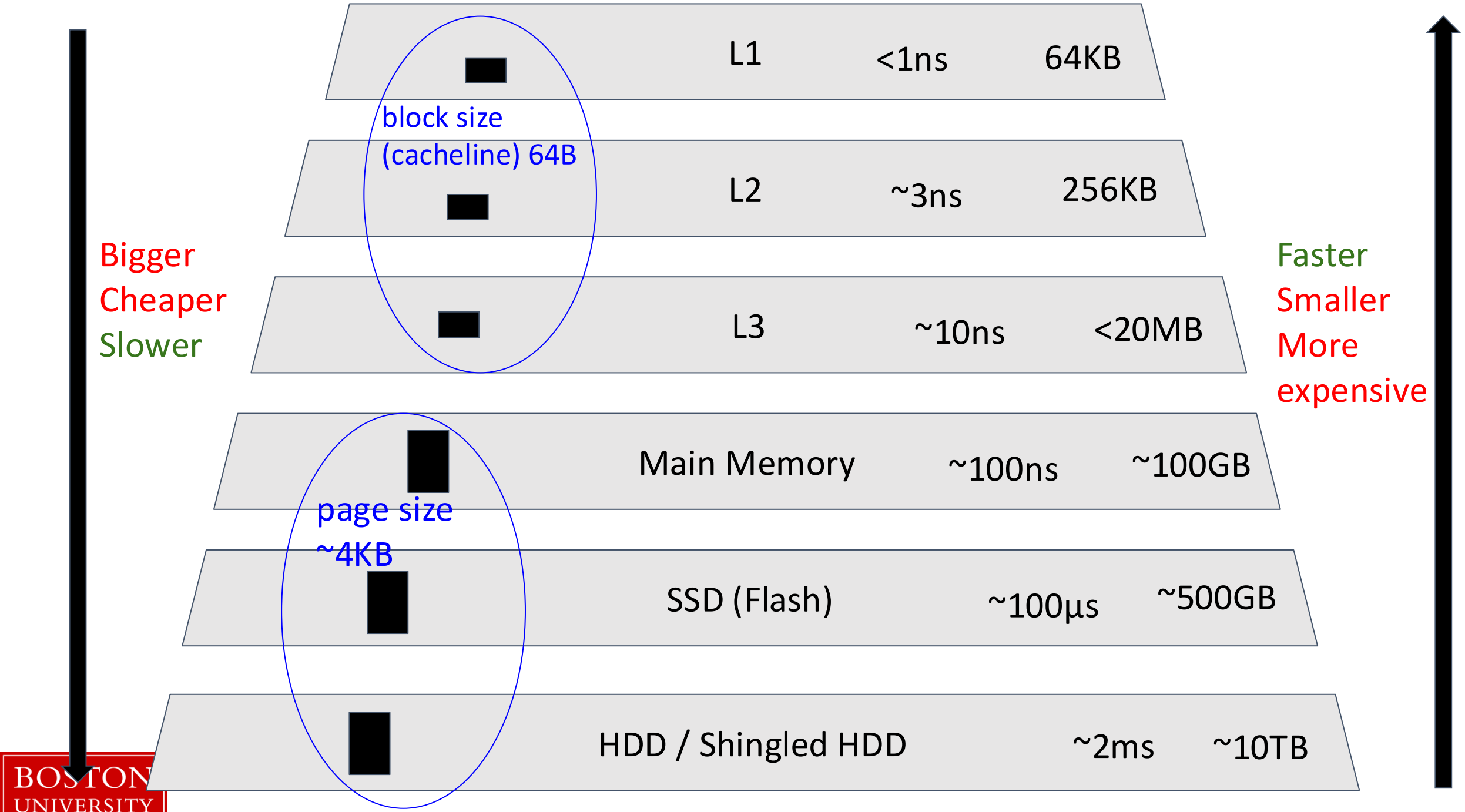


Reminder: Memory Wall





BOSTON UNIVERSITY



BOSTON
UNIVERSITY

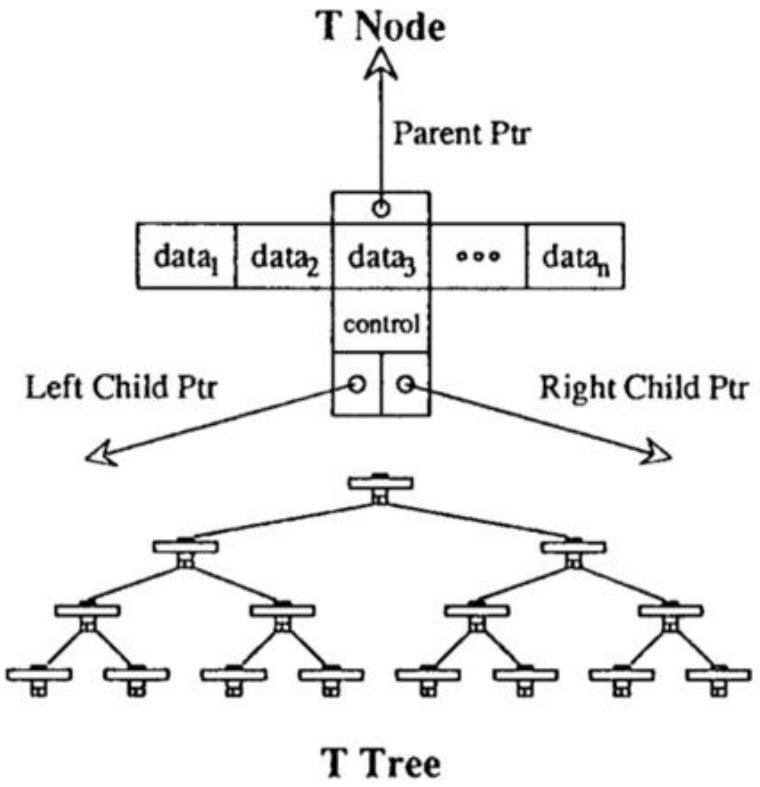
In-Memory Search Trees: T-Trees

Fat nodes (cacheline size) with **two** children

Developed in the 80s (still used in some systems!)

Unpredictable pointer chasing

Memory access **latency is not uniform**



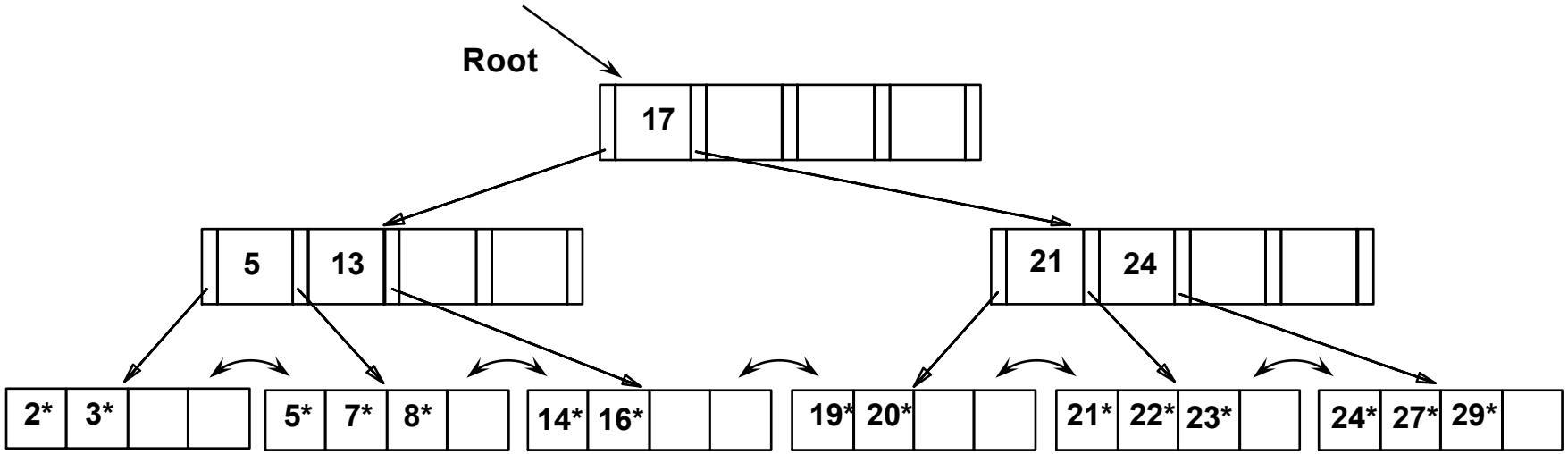
BOSTON UNIVERSITY

Are B+ Trees good for in-memory execution?

Designed for **disks**!

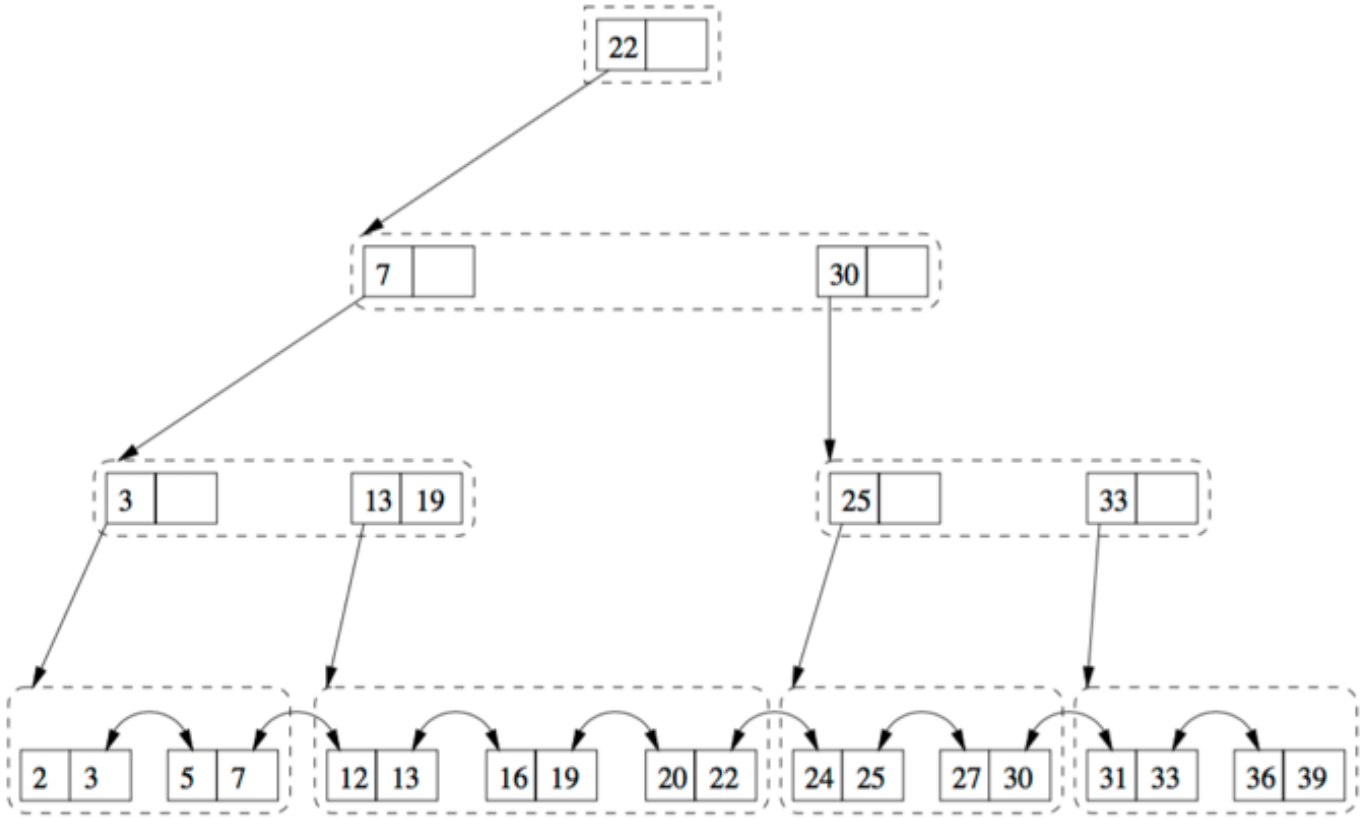
Node size is equal to **page size**, the goal is to *minimize #random accesses* of pages (wide fanout)

How to make it **memory-friendly**?



BOSTON UNIVERSITY

Cache-sensitive B+ Trees



Every level is physically stored **contiguously**

Good cache utilization!

Poor updates – needs logic to balance

why?

Tree **height** depends on **#items** inserted

similar to …?

BOSTON UNIVERSITY

Can we do better for an in-memory search tree?

Maintain order

tree

Maintain few random access

low height

Maintain good cache utilization

access cachelines

Maintain low space complexity

Cheap updates

less logic, avoid rebalancing or splitting

BOSTON
UNIVERSITY

Enter Tries

Also known as Radix Trees, Prefix Trees, Digital Trees

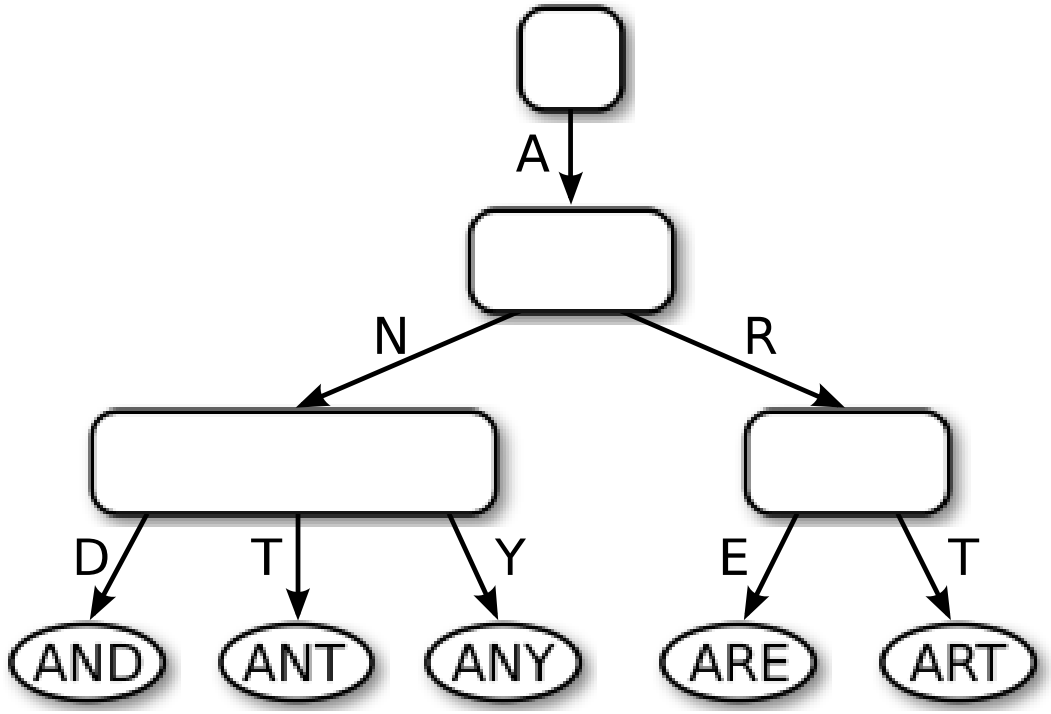
Trie, Radix Tree, Prefix Tree, Digital Tree

Tree **height** depends on **key length *k***

Not on tree (**data**) size

No rebalancing needed!

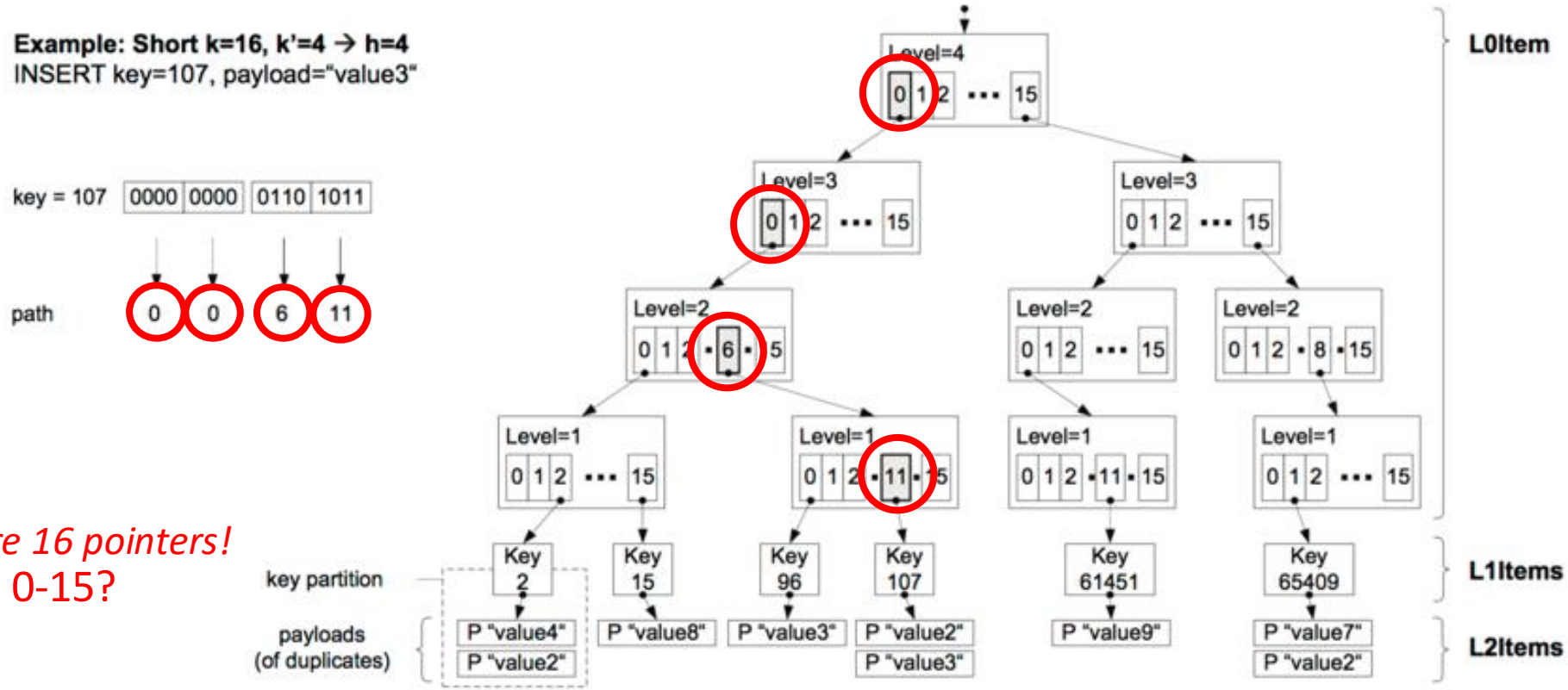
Automatically get lexicographical order



BOSTON UNIVERSITY

Tries on integers (in binary format)

Every node stores a part of the binary representation (“radix”) of the key



Implicit Keys

Just store 16 pointers!

Do you need to store values 0-15?

Significant **space** savings

Should all nodes use the **same number** of radix bits?

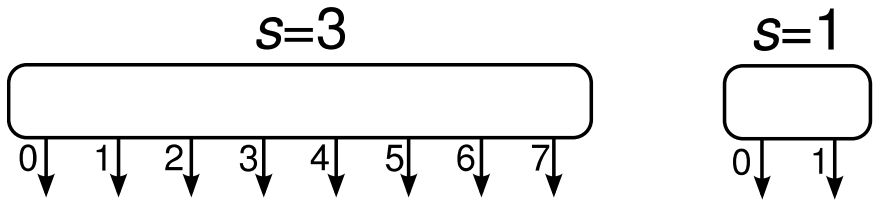
BOSTON UNIVERSITY

Adaptive Radix Tree Span

For **binary representations** of keys, the fanout can be configured!

Each node uses *s* bits ("span") of the radix of the key

Hence, an **inner node** is an array of 2^s pointers (with equal number of children)



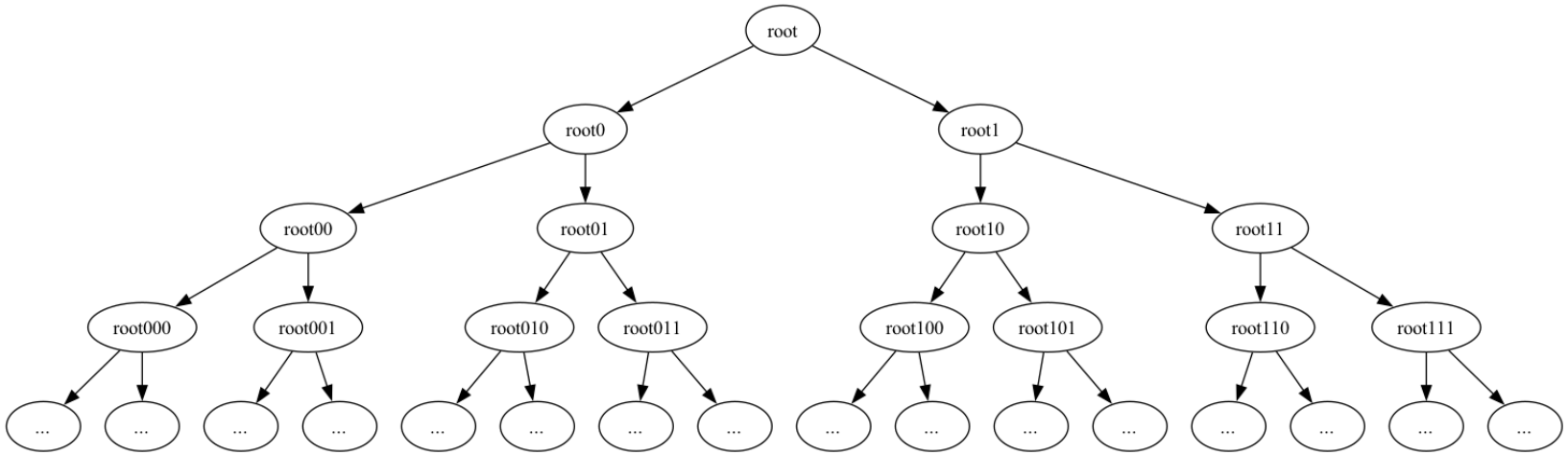
BOSTON UNIVERSITY

Tree Size vs. Span

k bit keys & span=s → k/s inner levels & 2^s pointers in each node

let's assume 32 bit keys

span=1 → 32 inner levels & 2 pointers in each node



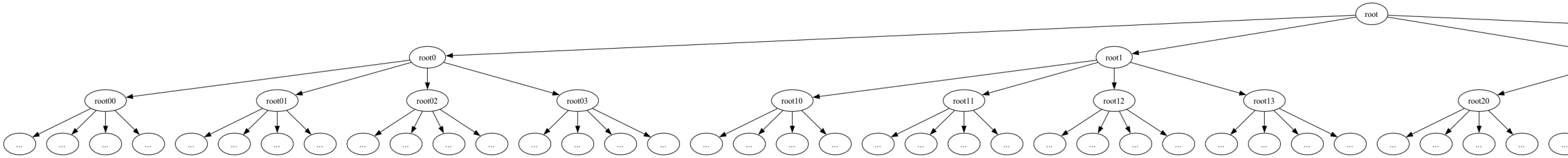
BOSTON UNIVERSITY

Tree Size vs. Span

k bit keys & span=s → k/s inner levels & 2^s pointers in each node

let's assume 32 bit keys

span=2 → 16 inner levels & 4 pointers in each node

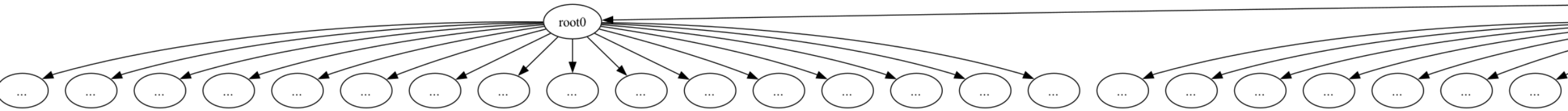


Tree Size vs. Span

k bit keys & span=s → k/s inner levels & 2^s pointers in each node

let's assume 32 bit keys

span=4 → 8 inner levels & 16 pointers in each node



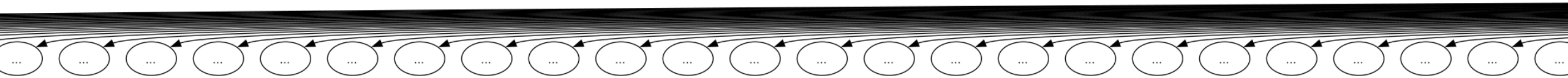
BOSTON UNIVERSITY

Tree Size vs. Span

k bit keys & span=s → k/s inner levels & 2^s pointers in each node

let's assume 32 bit keys

span=8 → 4 inner levels & 256 pointers in each node



BOSTON UNIVERSITY

Tree Size vs. Span

k bit keys & span=s → k/s inner levels & 2^s pointers in each node

let's assume 32 bit keys

span=1 → 32 inner levels & 2 pointers in each node tall and thin tree

span=2 → 16 inner levels & 4 pointers in each node

span=4 → 8 inner levels & 16 pointers in each node

span=8 → 4 inner levels & 256 pointers in each node short and fat tree

BOSTON UNIVERSITY

Height vs. Size Tradeoff

Large s:

small height (fast)

BUT

high space consumption

Small s:

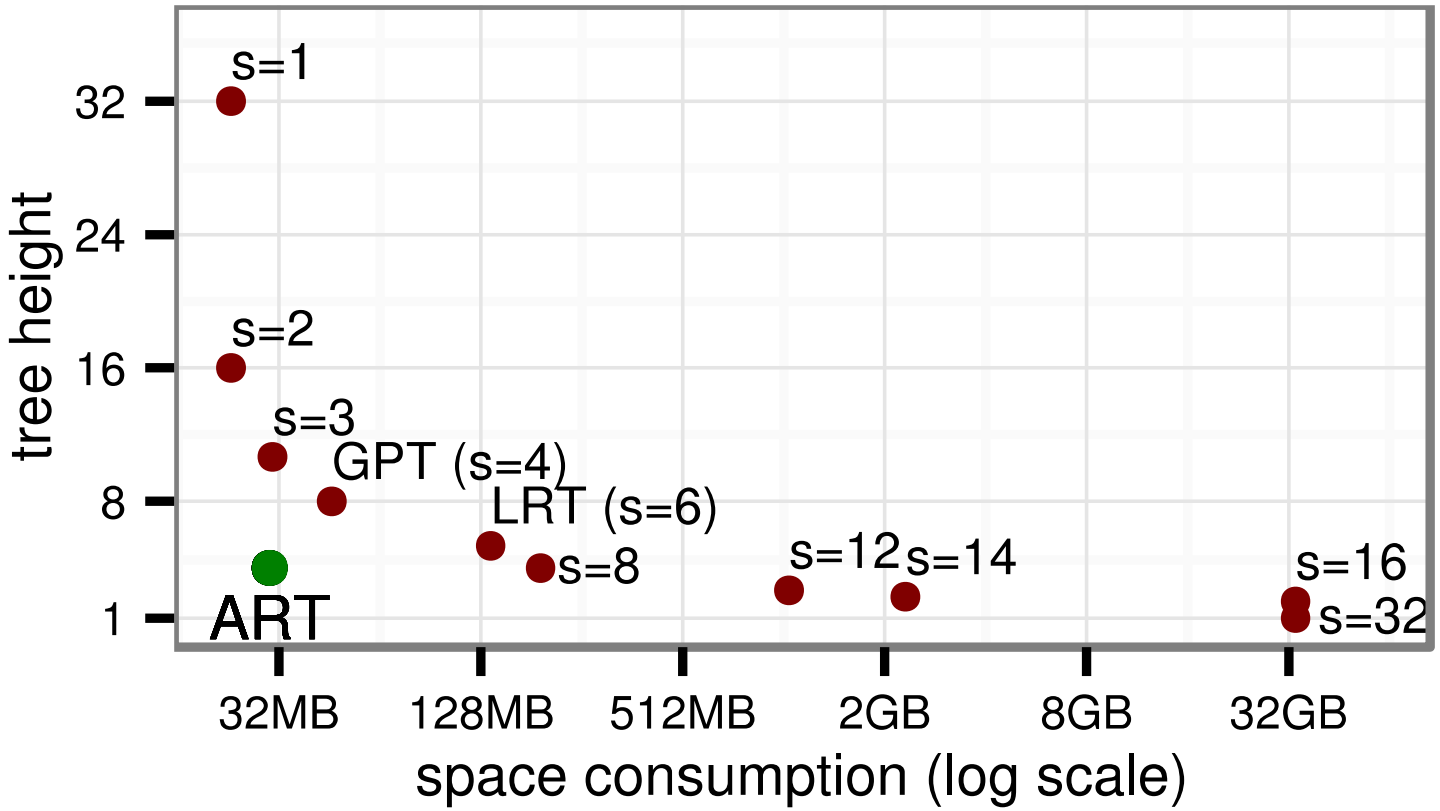
large height (slow)

BUT

low space consumption

ART manages to avoid this tradeoff

How?

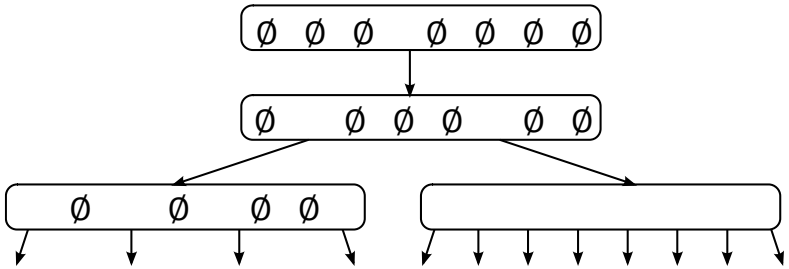


BOSTON UNIVERSITY

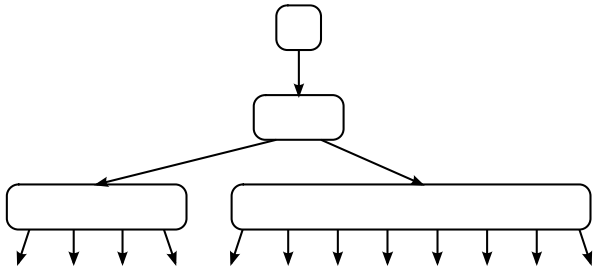
Adaptively Sized Nodes

s = 8: each inner node corresponds **1 byte of the key**

however: **different node sizes** depending on the **actual** number of children



a classical radix tree with fixed-side array nodes



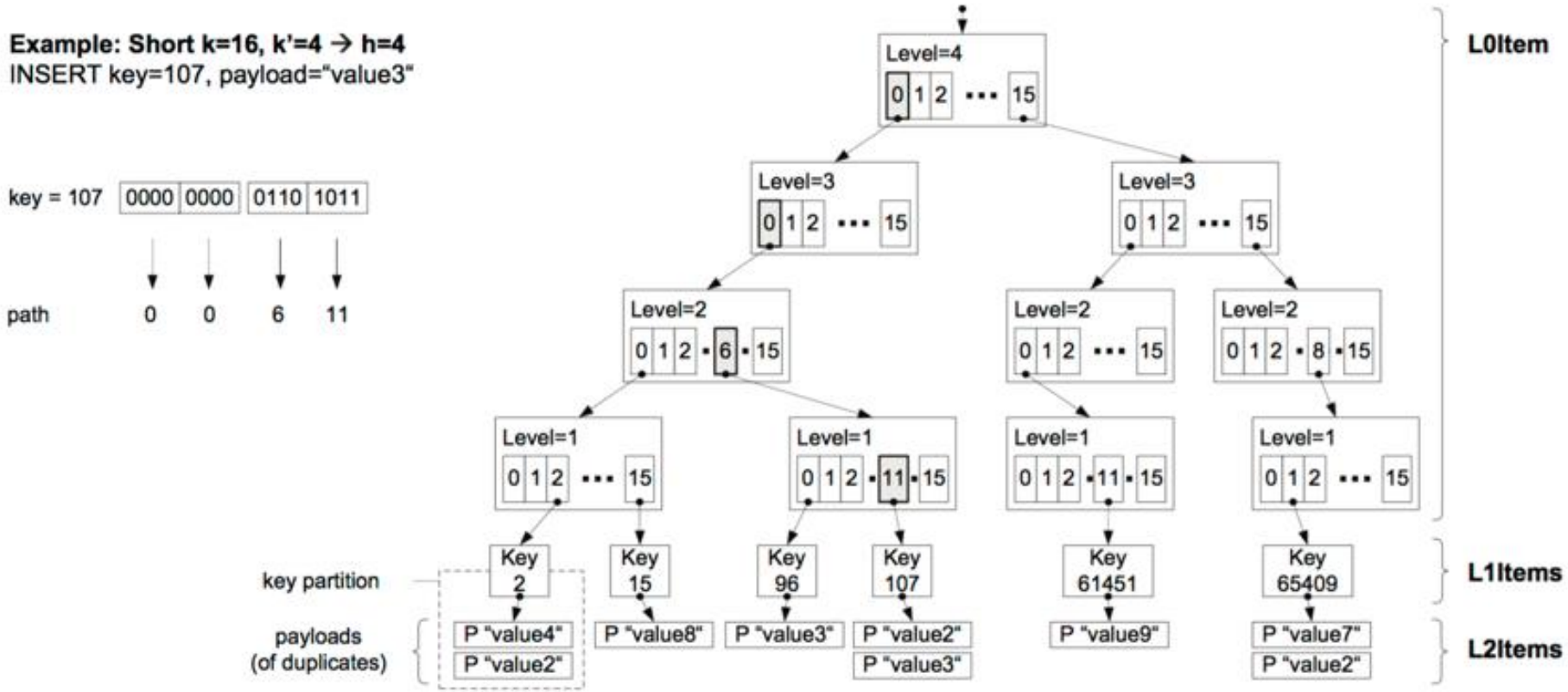
a radix tree with adaptively-sized nodes

design innovation: variable fanout

BOSTON UNIVERSITY

Remember: what is the goal?

to break a 32-bit key in 4 bytes across 4 levels and reduce the size of the nodes!



BOSTON UNIVERSITY

More on adaptive nodes

4 node sizes, dynamic decision

explicit keys
both Node4 and Node16
use arrays of size 16

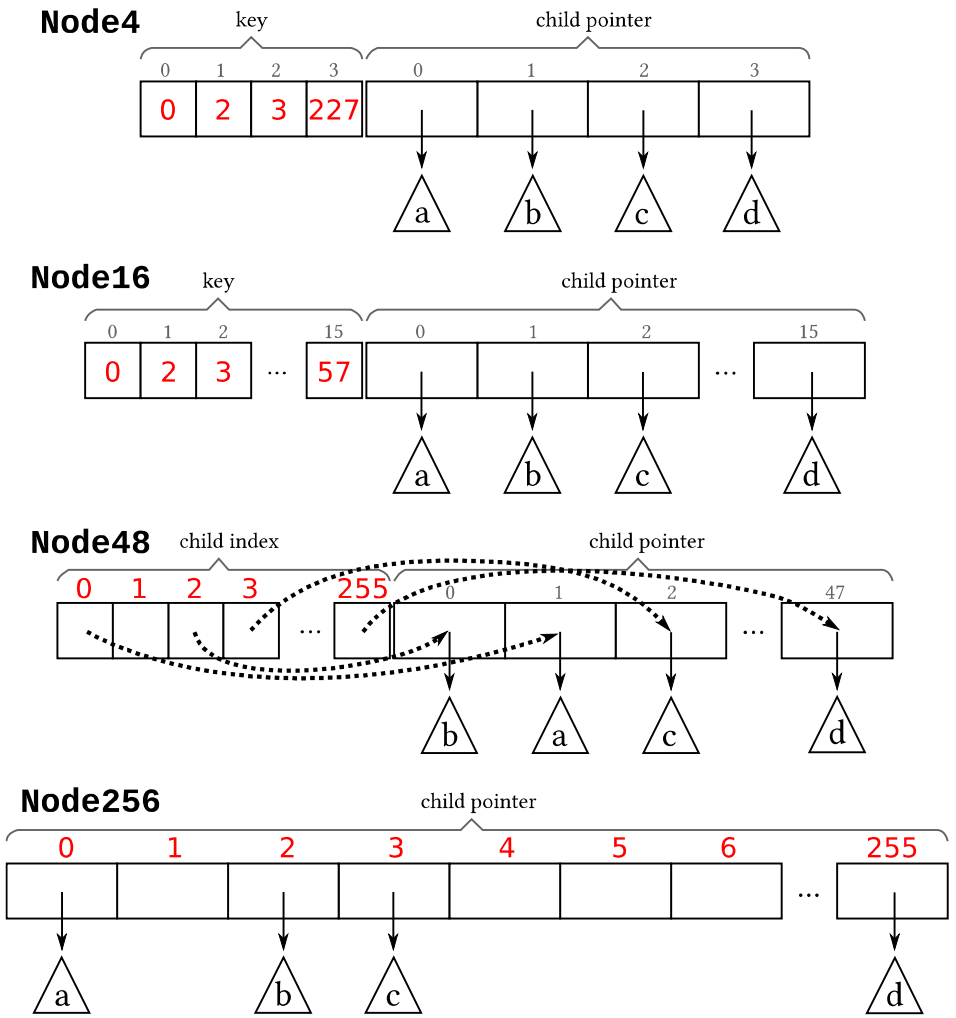
```
typedef struct {
    art_node n;
    unsigned char keys[16];
    art_node *children[16];
} art_node16;
```

indirection index
with implicit keys

```
typedef struct {
    art_node n;
    unsigned char keys[256];
    art_node *children[48];
} art_node48;
```

implicit keys

```
typedef struct {
    art_node n;
    art_node *children[256];
} art_node256;
```



BOSTON UNIVERSITY

More on adaptive nodes

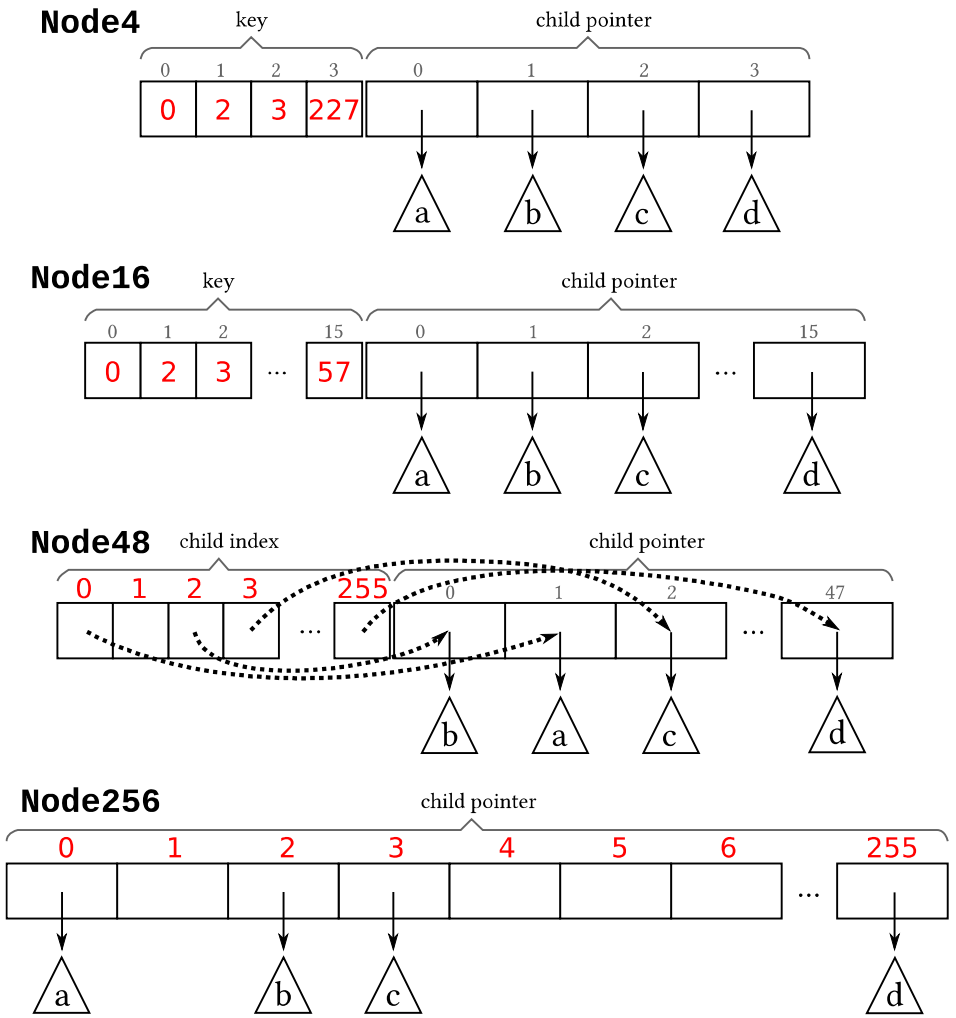
4 node sizes, dynamic decision

TABLE I
SUMMARY OF THE NODE TYPES (16 BYTE HEADER, 64 BIT POINTERS).

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	$16 + 256 \cdot 8 = 2064$

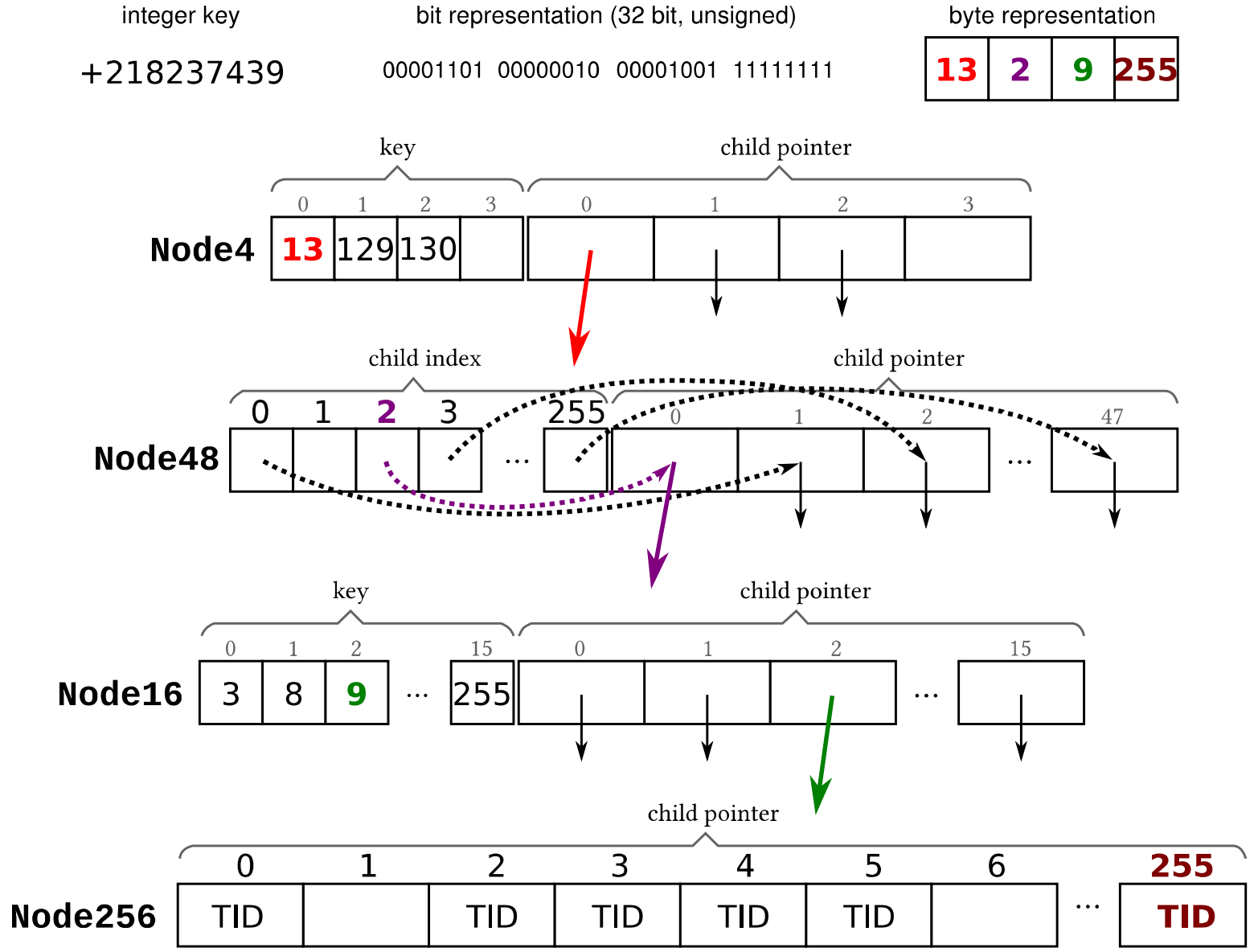
TABLE II
WORST-CASE SPACE CONSUMPTION PER KEY (IN BYTES) FOR DIFFERENT RADIX TREE VARIANTS WITH 64 BIT POINTERS.

	$k = 32$	$k \to \infty$
ART	43	52
GPT	256	∞
LRT	2048	∞
KISS	>4096	NA.



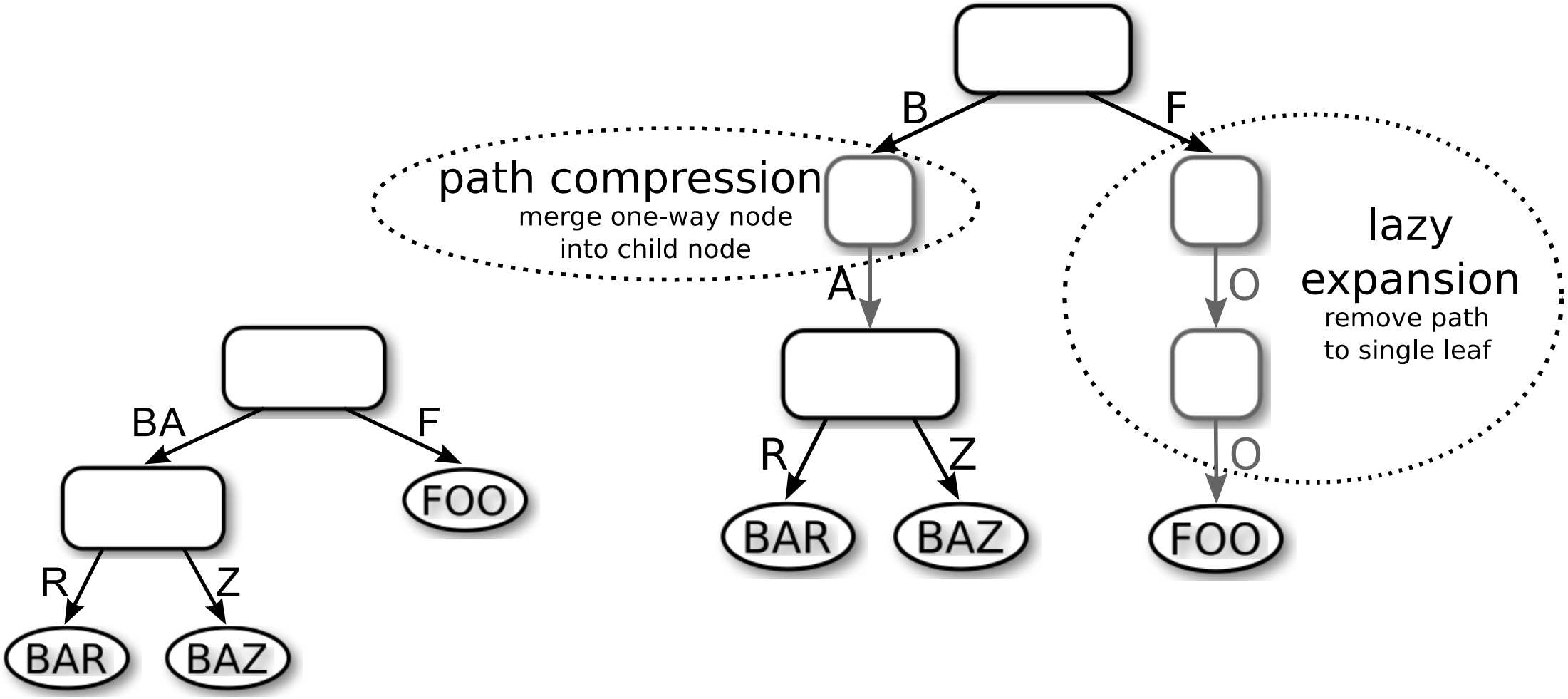
BOSTON UNIVERSITY

ART Traversal



BOSTON UNIVERSITY

Optimizations: Remove one-way nodes



BOSTON
UNIVERSITY

Supporting various data types

Native support for:

String

Integers (binary representation)

Require transformations for:

floats, Unicode, signed, null, composite

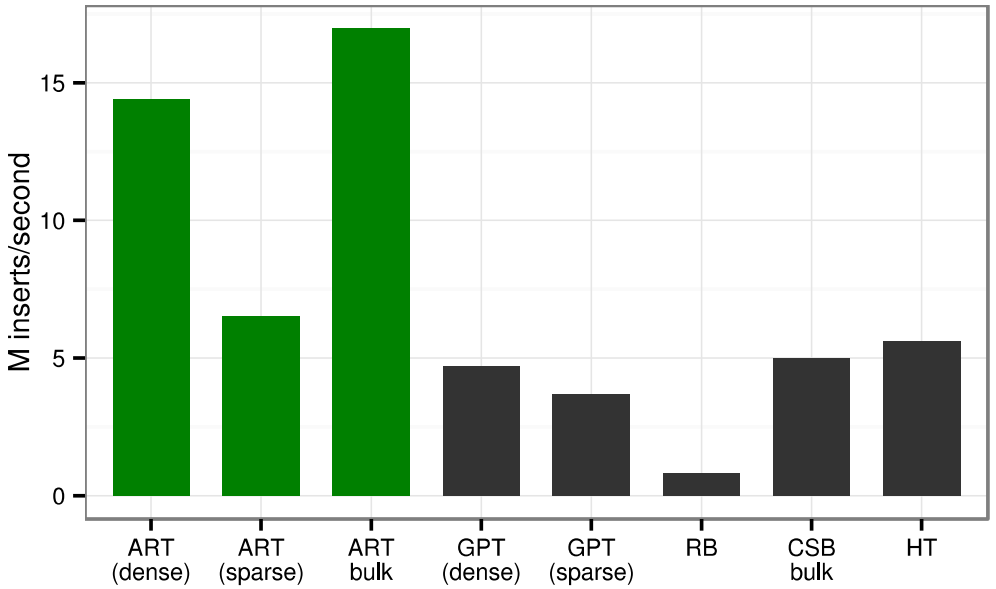
when?

depends on whether the bitwise order follows the intuitive order

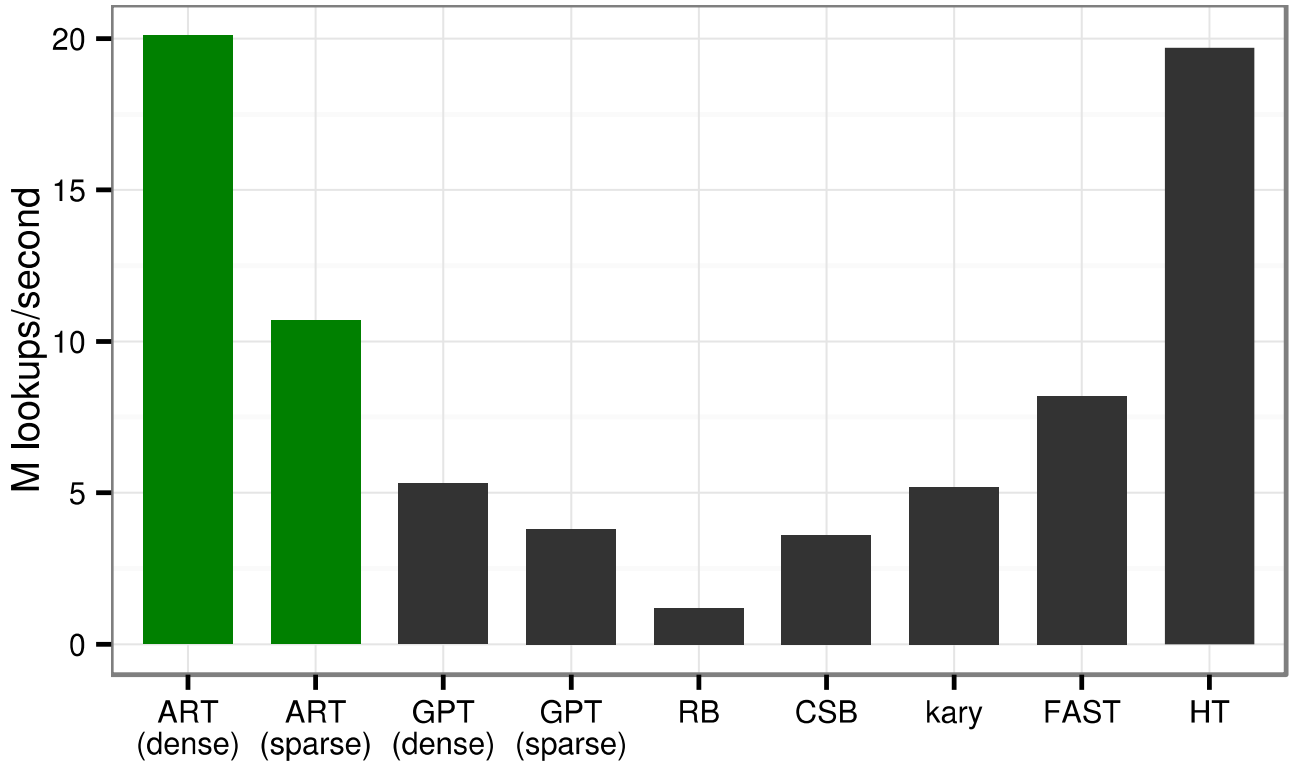
Example: for signed int negative values have their first bit as 1 (“greater” in terms of bitwise order)

Evaluation

Insert performance (4B keys)



Lookup performance (4B keys)



GPT: Generalized Prefix Tree, Boehm et al., BTW 2011

RB: Red-Black Tree

CSB: Cache-Sensitive B+Tree, Rao and Ross, SIGMOD 2000

kary: K-ary Search Tree, Schlegel et at., Damon 2009

FAST: Fast Architecture Sensitive Tree, Kim et al., SIGMOD 2010

HT: Chained Hash Table

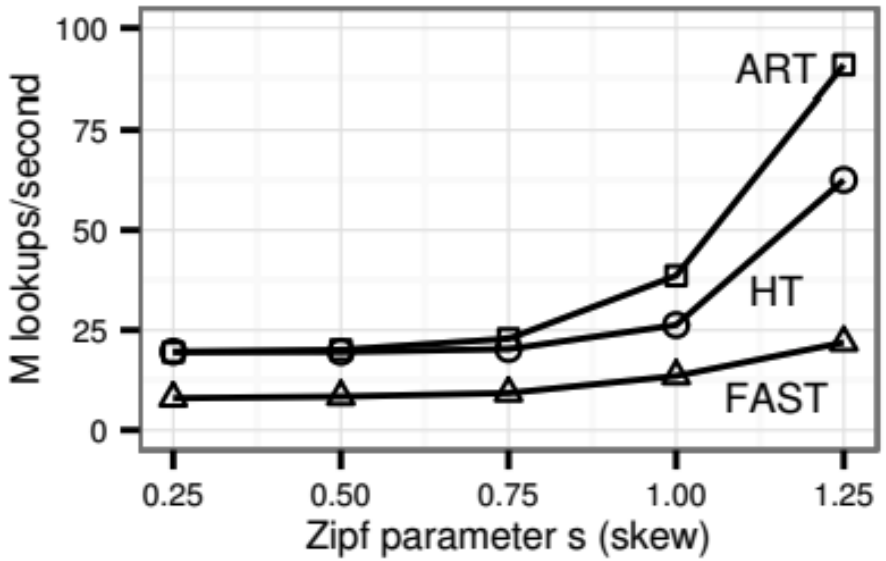
BOSTON UNIVERSITY

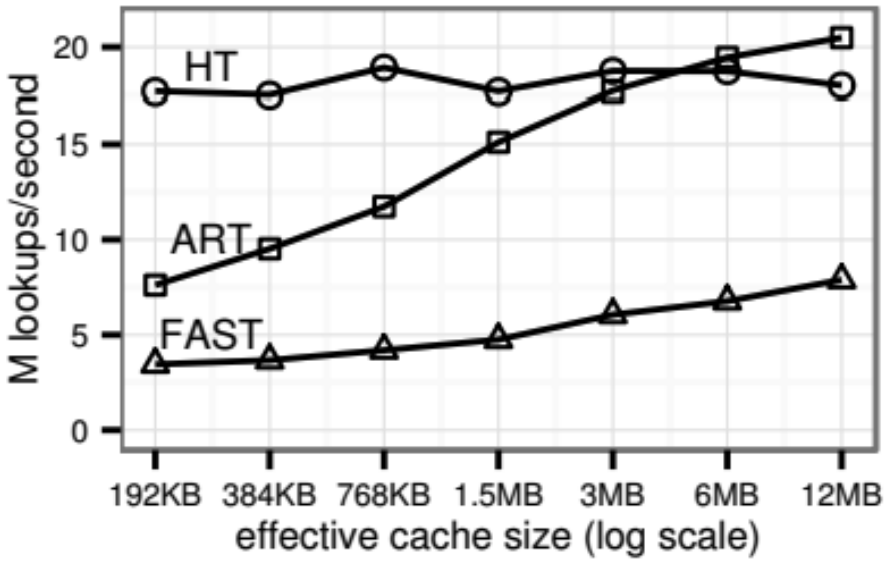
Cache Efficiency

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

Skewed Search & Impact of Cache Size





ART: adjacent items are in the same node/subtree

HT: adjacent items are in different buckets

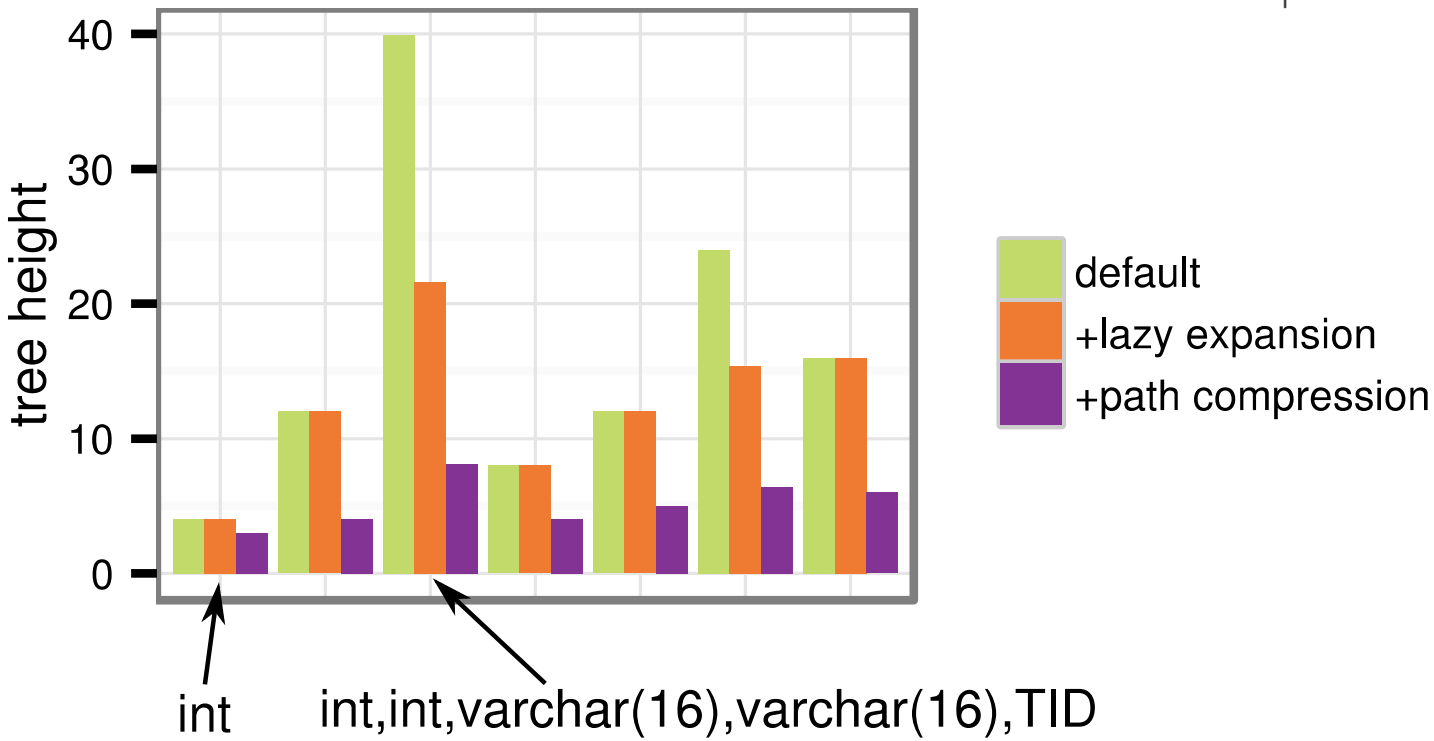
ART: no evictions, fewer misses overall

HT: data is randomly distributed more misses

BOSTON UNIVERSITY

Tree Height in TPCC

#	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



Without the height optimization the height can be the length of the keys → can be prohibitively high

why? what is the height of a B+ Tree?

BOSTON UNIVERSITY

Space Efficiency for TPCC

MAJOR TPC-C INDEXES AND SPACE CONSUMPTION PER KEY USING ART.

#	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

Conclusions

Radix Trees can be used as a **generalized** index

for multiple data types

space efficient

with excellent performance

thus, combining:

range query support of search trees

point lookup efficiency of hash indexes

BOSTON UNIVERSITY

BOSTON UNIVERSITY

CS 561: **Data Systems Architectures**

class 13

Adaptive Radix Trees

Prof. Manos Athanassoulis

https://bu-disc.github.io/CS561/