Dynamic RAM Price
Bits per Dollar at Production
(Packaged Dollars)

Logarithmic Plot

Doubling time: 1.5 years
Note that DRAM speeds have increased during this period.
Existing Solutions

Binary

Hash
Existing Solutions

Binary

\[ O(k \log n) \]

Hash

\[ O(k) \]
T tree

T Node

Parent Ptr

data_1  data_2  data_3  \ldots  data_n

control

Left Child Ptr

Right Child Ptr

T Tree
B+ Tree
Cache Aware B+ Tree
Radix tree - Implementation

K = 3
S = 1
|Alphabet| = 5
Radix trees vs B-trees

Radix trees:
- Height: k/s
- Complexity: O(k)

B-trees:
- Height: log n
- Complexity: O(k log n)
Cases when lexicographic order != desired order
Radix tree - Search

Value1  Value2

Find value for key: abc
Radix tree - Search

Search key (abc)

Time complexity O(K)
Insert key (abe, value3)
Radix tree - Insertion

Insert key (abe, value3)

Value1
Value2
Value3
Radix tree - Insertion

Insert key (ade, value4)
Radix tree - Insertion

Insert key (ade, value4)
Radix tree - Insertion

Lazy expansion
Lazy Expansion

Optimistic

Pessimistic
Improving space constraints

The diagram shows the relationship between tree height and space consumption (on a log scale) for different storage capacities (32MB to 32GB). The labels 's' and 'ART' indicate specific storage sizes and techniques, with 's' values ranging from 1 to 16. Points for GPT and LRT are also marked with 's' values of 4 and 6, respectively. This visual representation helps in understanding the trade-offs between storage size and space utilization efficiency.
Ideal Solution
Adaptive Radix trees

“The BMW of data structures”

- Kostas
Approximate Solution

Node 4

<table>
<thead>
<tr>
<th>key</th>
<th>child pointer</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td></td>
</tr>
<tr>
<td>255</td>
<td></td>
</tr>
</tbody>
</table>

Node 16

<table>
<thead>
<tr>
<th>key</th>
<th>child pointer</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td></td>
</tr>
<tr>
<td>255</td>
<td></td>
</tr>
</tbody>
</table>

...
Approximate Solution

16 + 4 + 4*8 = 52

16 + 16 + 16*8 = 160
Approximate Solution

Node 48

Node 256
Approximate Solution

16 + 256 + 48*8 = 656

16 + 256*8 = 2064
ART - insertion

Insert (afa)
ART - insertion

Insert (afa)
Insert (afg)
ART - insertion

Insert (afa)
Insert (afg)
Delete (afa)
ART - insertion

- Insert (afa)
- Insert (afg)
- Delete (afa)
Cache efficiency of ART?
Figure 1: (a) Node indices (=memory locations) of the binary tree (b) Rearranged nodes with SIMD blocking (c) Index tree blocked in three-level hierarchy – first-level page blocking, second-level cache line blocking, third-level SIMD blocking.
Point queries

<table>
<thead>
<tr>
<th></th>
<th>16M</th>
<th>256M</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>ART</td>
<td>ART</td>
</tr>
<tr>
<td></td>
<td>GPT</td>
<td>GPT</td>
</tr>
<tr>
<td></td>
<td>RB</td>
<td>RB</td>
</tr>
<tr>
<td></td>
<td>CSB</td>
<td>CSB</td>
</tr>
<tr>
<td></td>
<td>kary</td>
<td>kary</td>
</tr>
<tr>
<td></td>
<td>FAST</td>
<td>FAST</td>
</tr>
<tr>
<td></td>
<td>HT</td>
<td>HT</td>
</tr>
<tr>
<td></td>
<td>dense, sparse</td>
<td>dense, sparse</td>
</tr>
<tr>
<td></td>
<td>(GPT and CSB crashed)</td>
<td>(GPT and CSB crashed)</td>
</tr>
</tbody>
</table>

M lookups/second
Point queries

HT and ART perform better than all other data structures but ART performs worse for sparse set of keys.
Point queries

A comparison of adaptive radix trees and hash tables (ICDE 2015)
Better choices of hash functions outperform ART
Experiments

Fig. 11. Multi-threaded lookup throughput in an index with 16M keys (12 threads, software pipelining with 8 queries per thread).
Experiments

FAST shows significant improvement for multi-threaded lookup
ART performs worst for sparse keys

Fig. 11. Multi-threaded lookup throughput in an index with 16M keys (12 threads, software pipelining with 8 queries per thread).
Skewed key distribution

Fig. 12. Impact of skew on search performance (16M keys).
Skewed key distribution

Fig. 12. Impact of skew on search performance (16M keys).

Skewness is where ART does best: it adapts the node size and access speed accordingly.
Fig. 13. Impact of cache size on search performance (16M keys).
Fig. 13. Impact of cache size on search performance (16M keys).

ART performs better with larger cache size, but HT outperforms ART for smaller cache size.
Fig. 14. Insertion of 16M keys into an empty index structure.
ART is better because it does not need to rebalance the trees (it does need to update the node sizes)
Future steps

- Anticipate node types
- Learn which types are best from data
Fin