Basic External Memory Data Structures

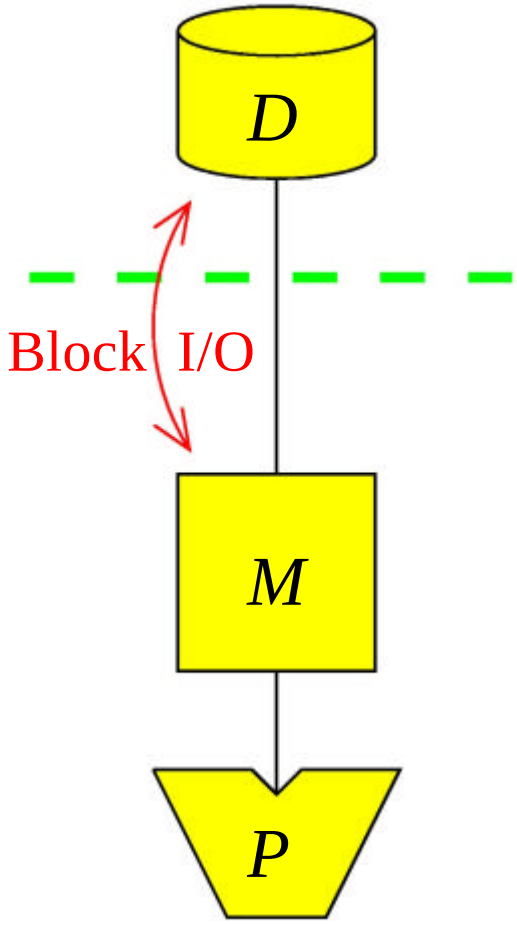
Rasmus Pagh

Aarhus University, Denmark

(Part of presentation prepared for ESA 2001 by Lars Arge, Duke University)

Dagstuhl, March 11, 2002

External Memory Model



In this talk, focus is on transfers between *two levels* of the memory hierarchy (I/O model)

N = # of items in the problem instance

B = # of items per external memory block

M = # of items that fit in internal memory

Z = # of items in output

We assume (for convenience) that $M > B^2$

Overview of Talk

Part I. Elementary data structures

- Data structures: Linked lists, stacks and queues
- Concepts: Locality (grouping), split-fuse, buffering

Part II. Search trees

- Data structures: B-trees, buffer trees
- Concepts: Balancing, output sensitivity, bootstrapping, batched dynamic problems

Part III. Non-comparison-based data structures.

- Data structures: Dictionaries, predecessor d.s., range dictionaries
- Concepts: Hashing, word length dependence

Part I. Elementary Data Structures

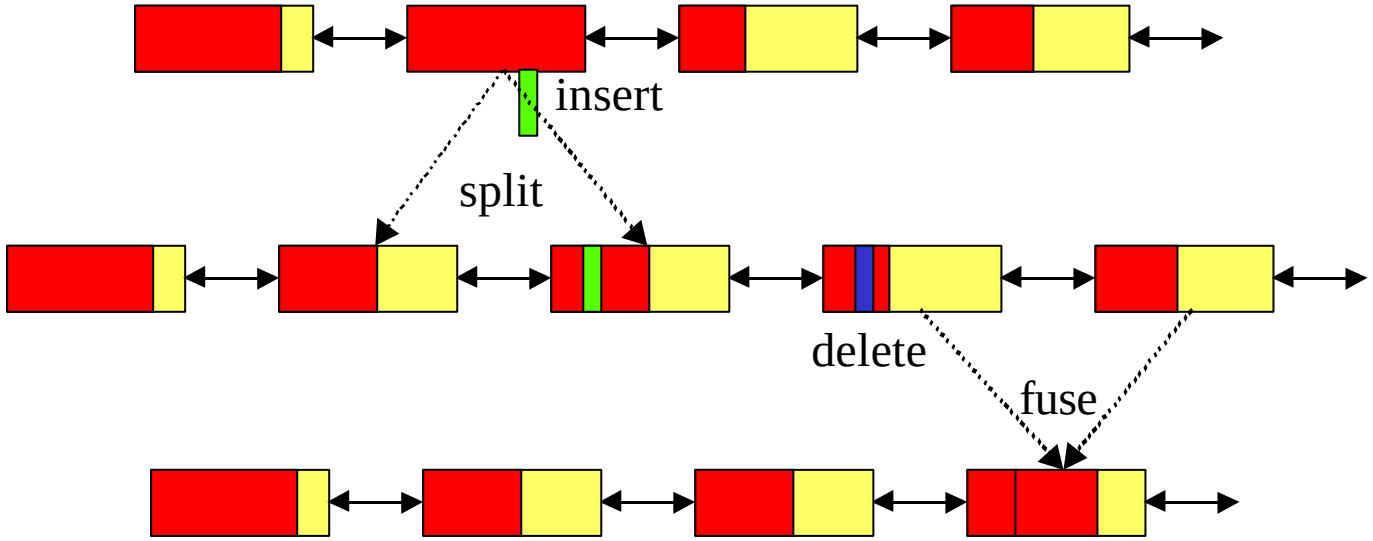
Linked Lists

- Usual internal memory data structures provide insertions and deletions in linked lists in *O(1)* I/Os
- Like in most internal pointer structures, there is no guarantee of *locality*, i.e., that data items needed around the same time resides close to each other in memory
- This means that traversal of Z elements may take *O(Z)* I/Os rather than the optimal bound of *O(1+Z/B)* I/Os for scanning a list

Optimal External Memory Linked Lists

Approach:

- To achieve locality, *group* adjacent elements in chunks of size $Q(B)$, stored in linked blocks
- Maintain chunk size by *splitting* and *fusing* chunks



Stacks and Queues

- Naïve externalization gives *O(1)* I/O per operation
- Much better performance can be achieved by *buffering*

Buffered queue:

- "Out-buffer" stores the $k£ B$ latest elements to enter the queue
- "In-buffer" stores the $k'£ B$ next elements to leave the queue
- Flushing out-buffer and filling in-buffer can be done in *1* I/O
  $\Rightarrow$ *O(1/B)* I/O amortized per operation

Buffered stack:

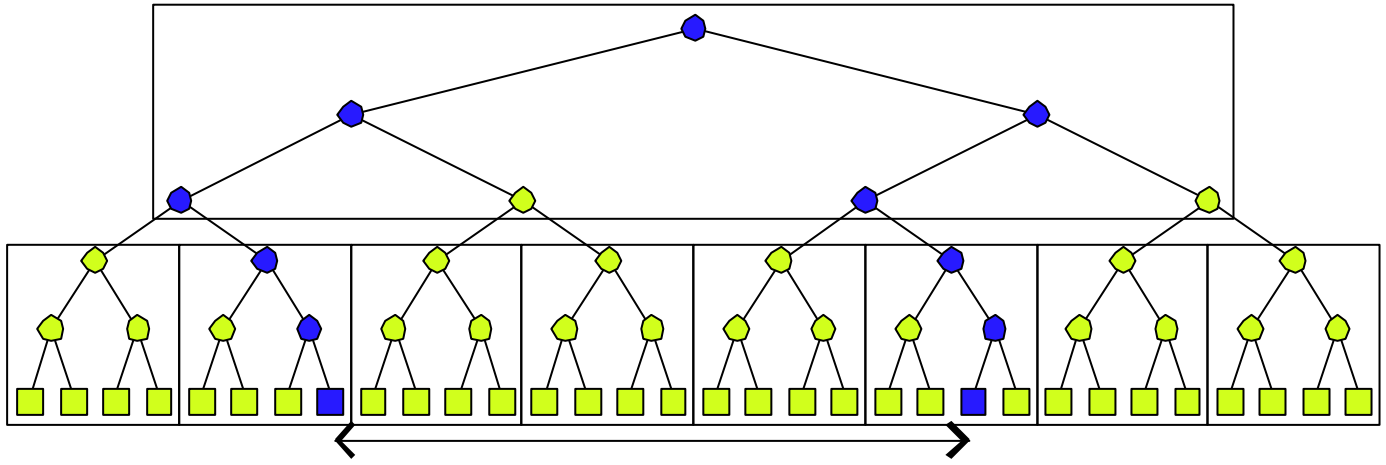
- Similar to the buffered queue
- *O(1/B)* I/O amortized per operation

Part II. Search Trees

B-trees

[Bayer-McCreight ‘72, Huddleston-Mehlhorn ‘82]

- Storing binary trees arbitrarily in external memory gives $O(\log_2 N)$ I/Os per query/update
- B-trees block nodes together such that each block can be stored in one external memory block
- The height of the tree (in blocks) becomes $O(\log_B N)$.



B-tree Summary

B-tree properties:

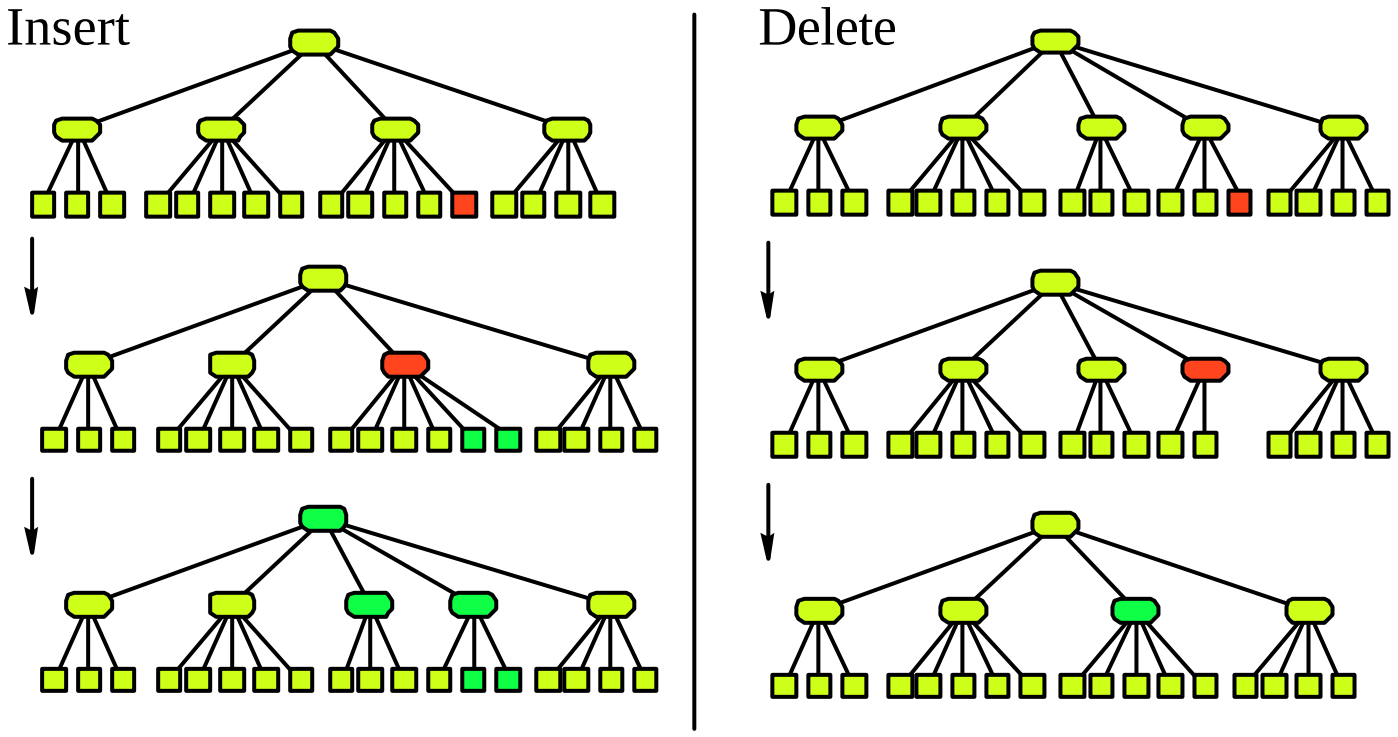
- All leaves – consisting of $\Theta(B)$ elements - on the same level
- Internal nodes have degree $\Theta(B)$ (except possibly the root)
- Depth is $O(\log_B N)$
- Uses $O(N/B)$ space (optimal)

Queries:

- Find a key, or the nearest neighbor of a key, in $O(\log_B N)$ I/Os
- Find all Z keys in a range $[a;b]$ (a "range search") in $O(\log_B N+Z/B)$ I/Os (output sensitive!)
- Time for queries is optimal for *comparison-based* algorithms.

Updates in B-trees

- Blocking hard to maintain by rotations as used in internal memory
- Rebalancing instead uses split/fuse (and share):



$\Rightarrow O(\log_B N)$ update bound

B-tree Applications

B-trees are used for a wide range of applications in computational geometry, graph algorithms, string processing, etc.

(many such applications to come in the next talks)

General goal:

- Search a database of N objects in $O(\log_B N+Z/B)$ I/Os
- Support updates in $O(\log_B N)$ I/Os

Output Sensitivity and Bootstrapping

Output sensitivity problem:

- Must be sure to report $W(B)$ elements for each I/O "after search".
- A *secondary structure* is often attached to each B-tree node to help reporting elements from subtrees with few elements to report
- When a node splits or fuses, the secondary structure must be rebuilt, which often involves traversing *the entire subtree*

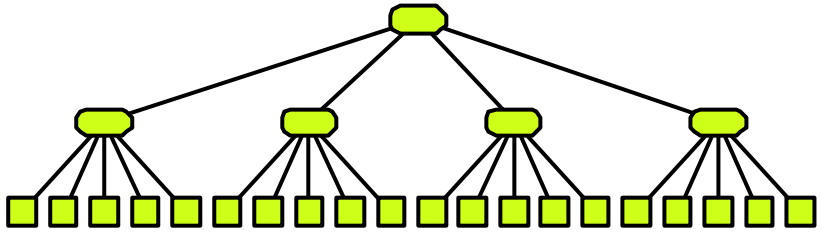
Bootstrapping:

- The secondary structure often solves a small version of the problem to be solved, i.e., it is used to *bootstrap* the main solution

Weight-balanced B-trees

Problem:

- Ordinary B-tree nodes can split/fuse often, so rebuilding the secondary structure can be expensive.



Weight-balanced B-tree:

- Like B-tree but with *weight* instead of degree constraint
- Balanced with split/fuse as B-tree
  - node ν only splits/fuses for every $\Omega(\omega(\nu))$ updates below it
  - amortized $O(\log_B N)$ I/O update if the secondary structure can be rebuilt in $O(\omega(\nu))$ I/Os

Persistent B-trees

- In database applications we are often interested in being able to access previous versions of database
- Partial persistence:
  - Update current version, query all versions
- Partially persistent B-tree (multi-version B-tree) can be obtained using standard techniques [DSST '89, BGOSW '96, VV '97]
  - $O(\log_B N)$ update, $O(\log_B N+Z/B)$ query, $O(N/B)$ space
  - N is total number of operations performed
- Idea:
  - Elements and nodes augmented with existence intervals
  - Maintain that every node contains $\Theta(B)$ alive elements in its existence interval

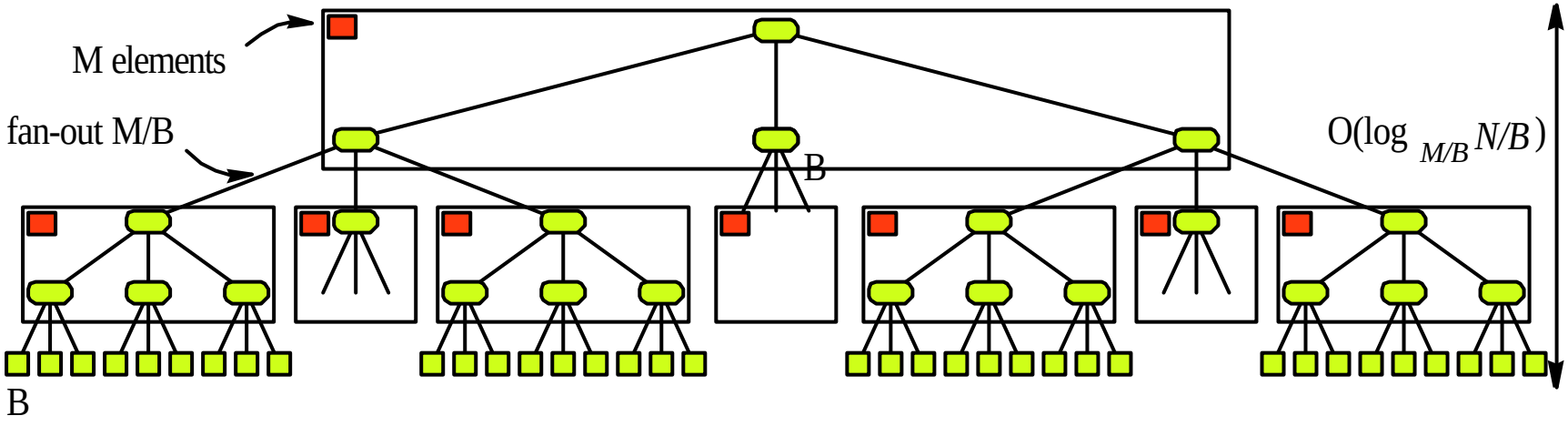
B-tree Construction

- Insertion in $O(\log_B N)$ I/Os in B-tree, weight-balanced B-tree, and persistent B-tree
- Sorting N elements takes $\Theta(\frac{N}{B}\log_{M/B}\frac{N}{B})$ I/Os
- Construction by repeated insertion use $O(N\log_B N)$ I/Os
  - more than a factor B sub-optimal

- We can build B-tree and weight-balanced B-tree in $\Theta(\frac{N}{B}\log_{M/B}\frac{N}{B})$ I/Os bottom-up
  - not so clear with persistent B-tree
- We need $O(\frac{\log_{M/B}\frac{N}{B}}{B})$ I/Os per operation to be optimal

Buffer Tree

[Arge '95]



- Main idea: Logically group nodes together and add buffers
  - Insertions done in a "lazy" way — elements inserted in buffers.
  - When a buffer runs full elements are pushed one level down.
  - Buffer-emptying in $O(M/B)$ I/Os

    $\Rightarrow$ every *block* touched constant number of times on each level

    $\Rightarrow$ inserting N elements (N/B blocks) costs $O(\frac{M}{B} \log_{M/B} \frac{N}{B})$ I/Os.

Buffer Tree Summary

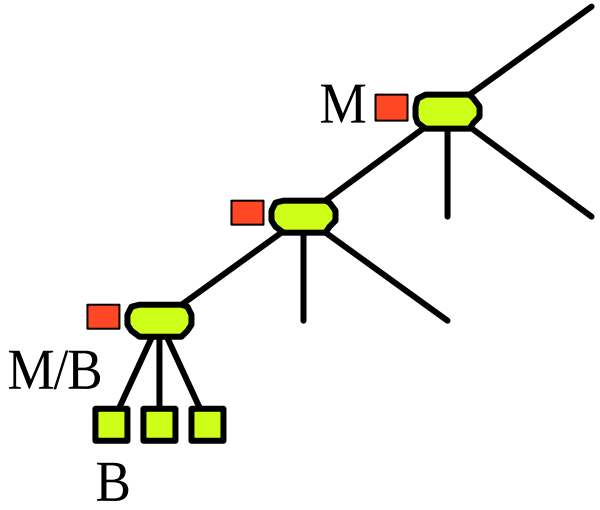
- Buffer tree can be used in $O(\frac{N}{B}\log_{M/B}\frac{N}{B})$ sorting algorithm
- Deletes and range queries can be handled similarly to insertions
  - Queries become *batched*, i.e., answers are not given right away.
- Batched dynamic problems occur in many applications, in particular "sweepline" algorithms in computational geometry.

Buffer Tree Priority Queue

Buffer trees also yield an optimal comparison-based priority queue.

Approach:

- Empty all buffers on leftmost path
  $\Rightarrow O(\frac{M}{B} \log_{M/B} \frac{N}{B})$ I/Os
- Delete M elements in leftmost leaves
- Next M deletemins free $\Rightarrow O(\frac{\log_{M/B} \frac{N}{B}}{B})$ I/Os amortized



A "worst-case efficient" priority queue also exists [BK '98].

Implementations

Libraries:

- LEDA-SM (www.mpi-sb.de/~crauser/leda-sm)
- TPIE (www.cs.duke.edu/TPIE)

Implementations of basic external memory data structures:

- B-trees — LEDA-SM, TPIE
- Persistent B-trees — TPIE
- Buffer trees — LEDA-SM
- Priority queues — LEDA-SM, TPIE

Part III. Non-comparison-based Data Structures

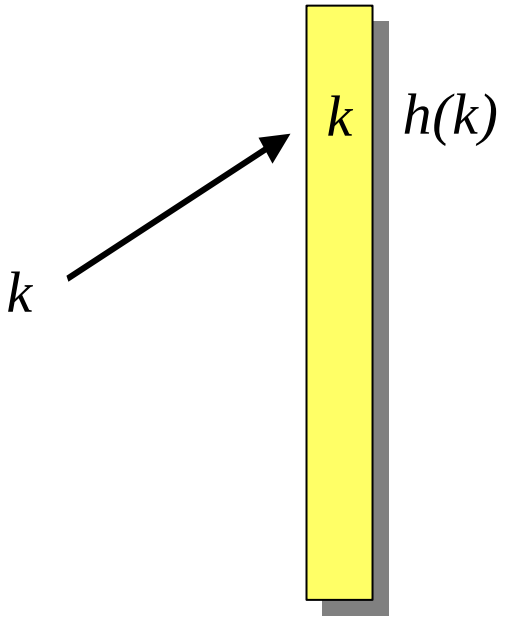
Dictionaries and Classic Hashing

Hashing in a nutshell:

- Use a function *h* to guide searches
- Search for key *k* (starting) at position *h(k)* in a ”hash table”

Intuition:

If *h* behaves ”randomly”, it distributes the keys evenly throughout the table



Collision resolution:

– Mechanism for dealing with keys hashing to the same location

– Easier* in external memory, as each block can accommodate *B* keys.

External Performance of Classic Hashing

Let α = *load factor* (fraction of hash table occupied by keys)

- Hash table consists of $N/(B\alpha)$ blocks
- The probability that a given block overflows is at most

$$e^{-B(1/\text{á}-1)^2/3}$$

- *No block overflows* unless N is exponentially larger than B (w.h.p.)

Alternative collision resolution:

– Store elements from overflowing blocks in internal memory.

– Guaranteed 1 I/O dictionary lookup

Two-way Hashing

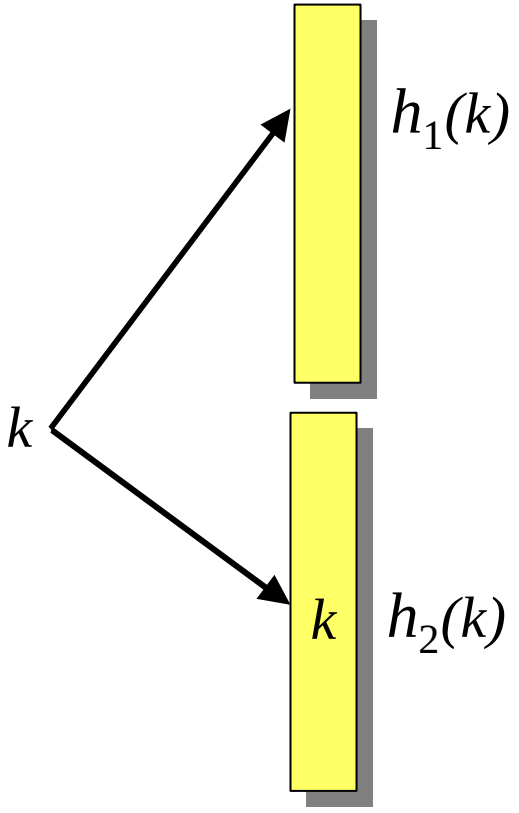
[ABKU '99, BCSV '00]

If two "banks" of external memory can be accessed simultaneously, better "1 I/O" probability bounds can be achieved

Two-way Hashing:

- Use *two* hash functions h_1 and h_2 with disjoint ranges
- Insert new key k in the *least full* of block $h_1(k)$ and $h_2(k)$
- The probability that a given block overflows is:

$$2^{-2^{\Omega((1-\alpha)B)}}$$



Resizing Hash Tables

Maintaining a (high) load factor:

- When the size of the key set changes, so must the hash table size
- Rehashing should be easy, ideally done in a single linear scan

Using a ”mother” hash function:

- Use hash function φ with large range to derive other hash functions
- Poor choice: $x \mapsto \varphi(x) \bmod r$
- Good choice: $x \mapsto \varphi(x) \operatorname{div} (R \operatorname{div} r)$, where R = size of φ’s range
  - The ordering of hash function values never changes

Other Non-comparison-based Data Structures

Dictionary generalizations:

- **Predecessor dictionary:**
  Given k find the largest key j in the dictionary with $j<k$ (if any)
- **Range dictionary:**
  Given a, b find all keys j in the dictionary with $j \in [a\ ;\ b]$

Techniques:

- Use and manipulate bit representation of keys (w bits)
- Recurse on smaller set *or shorter keys*
- Use dictionaries to look up ”pre-computed” answers

Predecessor Dictionary

There is a predecessor dictionary [A'96,BF'99,AT'00] that uses linear space and, for N keys of w bits supports updates and queries in time

$$O\left(\min\left(\sqrt{\log N / \log\log N}, \log_{Bw} N, \frac{\log w}{\log\log w}\log\log_{Bw} N\right)\right)$$

Any predecessor dictionary using space poly(n) must have query time

$$\Omega\left(\min\left(\log_{Bw} N, \frac{\log w}{\log\log w}\right)\right) \qquad \text{[BF '99]}$$

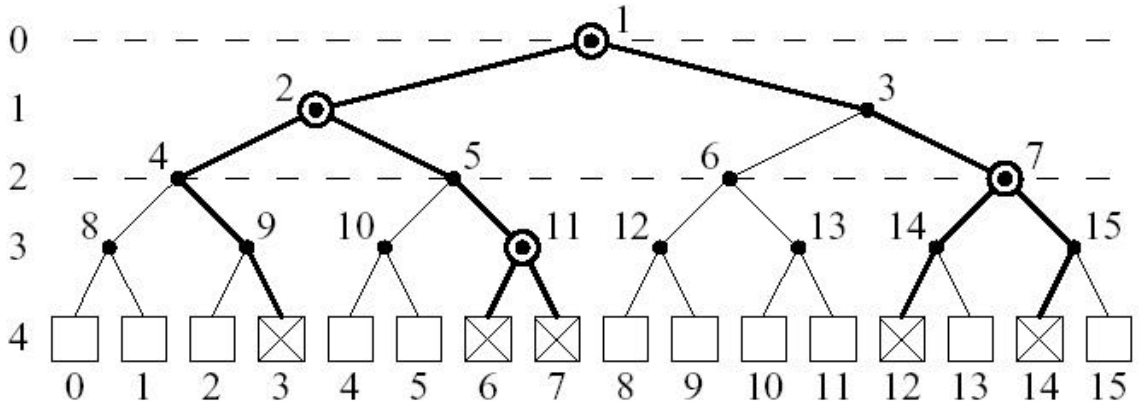
The best possible query time in terms of N is:

$$\Omega\left(\min\left(\sqrt{\log N / \log\log N}, \log_B N\right)\right)$$

Range Dictionary

Main observations:

- We report only keys that share the longest common prefix of *a* and *b*
- The trie of *N* *w*-bit strings can be stored such that the longest prefix of any string can be found in constant time. Main issue: Space



Result:

- Range queries in $O(1+Z/B)$ I/Os and space $O(N)$
- Static, no efficient dynamization known (yet)

Summary

Part I. Elementary data structures

- Data structures: Linked lists, stacks and queues
- Concepts: Locality (grouping), split-fuse, buffering

Part II. Search trees

- Data structures: B-trees, buffer trees
- Concepts: Balancing, output sensitivity, bootstrapping, batched dynamic problems

Part III. Non-comparison-based data structures.

- Data structures: Dictionaries, predecessor and range dictionaries
- Concepts: Hashing, word length dependence