CS 240 – Data Structures and Data Management

Module 11: External Memory

A. Storjohann
Based on lecture notes by many previous cs240 instructors

David R. Cheriton School of Computer Science, University of Waterloo

Fall 2018

References: Goodrich & Tamassia 14.1, Sedgewick 16.4

version 2018–11–27 10:33

Outline

1. External Memory
   - Motivation
   - External sorting
   - External Dictionaries
   - 2-3 Trees
   - (a, b)-Trees
   - B-Trees
   - Extendible Hashing

Outline

Different levels of memory

Current architectures:

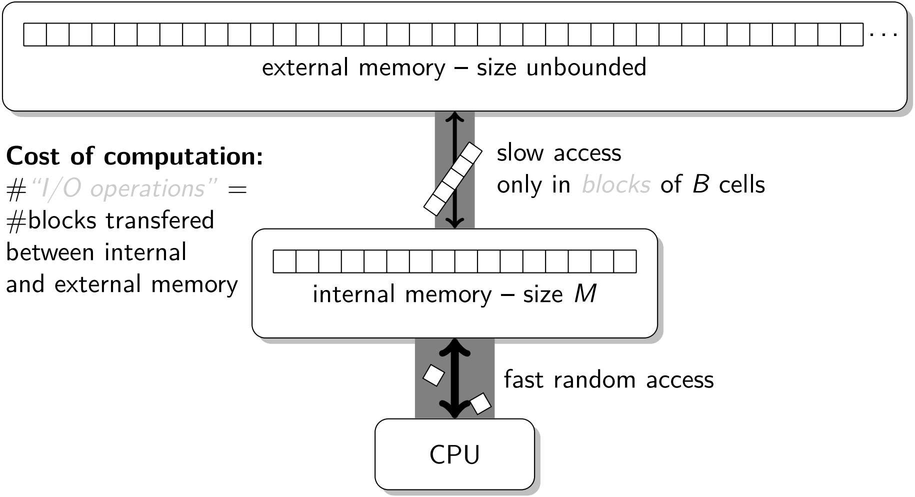
- registers (very fast, very small)
- cache L1, L2 (still fast, less small)
- main memory
- external memory: disk or cloud (slow, very large)

General question: how to adapt our algorithms to take the memory hierarchy into account, avoiding transfers as much as possible?

Observation: Accessing a single location in *external memory* (e.g. hard disk) automatically loads a whole block (or "page").

New objective: revisit all ADTs/problems with the objective of minimizing page loads.

The External-Memory Model (EMM)



Cost of computation:
#*"I/O operations"* =
#blocks transfered
between internal
and external memory

I/O-operations are also called *page loads* or *block transfers*.

Outline

1. External Memory
   - Motivation
   - External sorting
   - External Dictionaries
   - 2-3 Trees
   - (a, b)-Trees
   - B-Trees
   - Extendible Hashing

Sorting in external memory

Given an array A of n numbers, put them into sorted order.

Now assume n is huge and A is stored in blocks in external memory.

- Recall: Heapsort was optimal in time and space in RAM model
- **But:** Heapsort accesses A at indices that are far apart
  $\rightsquigarrow$ typically one page loads per array access.
- Mergesort adapts well to an array stored in external memory.
- It can be made even more effective using **d-way merge**: Merge d sorted runs into one sorted run.

d-way merge

d-Way-Merge$(S_1, \ldots, S_d)$
$S_1, \ldots, S_d$ are sorted sets (arrays/lists/stacks/queues)

```
P ← empty min-priority queue
S ← empty set
for i ← 1 to d do
    P.insert((first element of S_i, i))
while P is not empty do
    (x, i) ← deleteMin(P)
    remove x from S_i and append it to S
    if S_i is not empty do
        P.insert((first element of S_i, i))
```

- Standard mergesort uses $d = 2$
- $d > 2$ could be used in internal memory as well, but the extra time to find minimum in the priority queue means the overall run-time is no better.

Mergesort in external memory

External ($B = 2$):

39	5	28	22	10	33	29	37	8	30	54	40	31	52	21	45	35	11	42	53	13	12	49	36	4	14	27	9	44	3	32	15	43	2	17	6	46	23	20	1	24	7	18	47	26	16	48	50

Internal ($M = 8$):

1. Create n/M sorted runs of length M. $\Theta(n/B)$ **IO-operations**
2. Merge the first $d \approx M/B - 1$ sorted runs using *d-Way-Merge*
3. Keep merging the next runs to reduce # runs by factor of d $\rightsquigarrow$ one round of merging. $\Theta(n/B)$ **IO-operations**
4. $\log_d(n/M)$ **rounds** of merging create sorted array.

Mergesort with external memory

Total page loads: $O(\log_d(n) \cdot n/B)$.

Assuming the EMM, one can prove lower bounds!

- $\Omega(\frac{n}{B})$ I/Os required to **scan** n elements.
- $\Omega(\frac{n}{B} \log_{M/B}(\frac{n}{B}))$ I/Os required to **sort** n elements with comparisons.
  - We don't prove that here.
- d-way Mergesort with $d \approx M/B$ is optimal (up to constant factors)!

Outline

Dictionaries in external memory

Tree-based dictionary implementations have poor *memory locality*:
If an operation accesses m nodes, then it must access m spaced-out memory locations.

- In an AVL tree, $\Theta(\log n)$ pages are loaded in the worst case.
- Better solution: do more in single node $\rightsquigarrow$ B-trees

- First consider special case of B-trees: *2-3 trees*
  - 2-3-trees would also be interesting for implementing ADT Dictionaries in main memory (may be even faster than AVL-trees)
  - We first analyze their performance in main memory, and then (for B-trees) in external memory.

Outline

1. External Memory
   - Motivation
   - External sorting
   - External Dictionaries
   - 2-3 Trees
   - (a, b)-Trees
   - B-Trees
   - Extendible Hashing

2-3 Trees

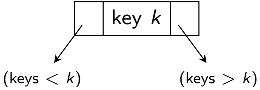
A 2-3 Tree is a balanced search tree that is not necessarily binary.

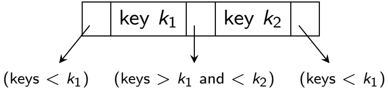
Structural properties:

- Every internal node is either
  - 1-node: *one KVP* and *two children*, or
  - 2-node: *two KVPs* and *three children*.
- The external nodes are *NIL* (do not store keys)
- All external nodes are at the same level.

Height-balance strictly enforced, but allow 2 types of nodes!

Order property: The keys at a node are between the keys in the subtrees.





2-3 Tree operations

Search: The order-property determines the subtree to search in.

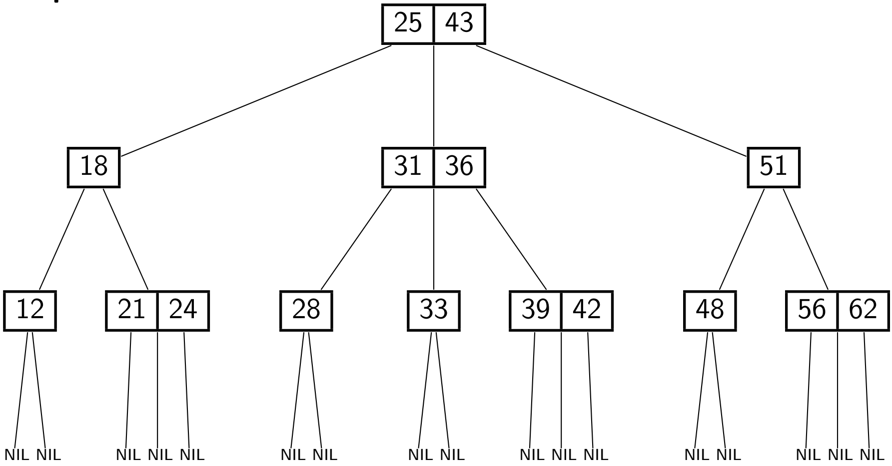
```
23TreeSearch(k, v ← root)
    Let c_0, k_1, ..., k_d, c_d be keys and children at v, in order
    if k ≥ k_1
        i ← maximal index such that k_i ≤ k
        if k_i = k return k_i
    else i ← 0
    23TreeSearch(k, c_i)
```

Insert: Nodes may grow from bottom to top.

- Search to find leaf ℓ where the new key k belongs.
- Add k and a NIL-child to ℓ. If ℓ now as 3 keys (**overflow**):
  - Split ℓ into two nodes ℓ, ℓ' with min and max key of ℓ
  - Move median key of ℓ into parent p of ℓ. Also make ℓ' child of p.
  - Recurse in p if it now has overflow.

Example: Insertion in a 2-3 tree

Example:

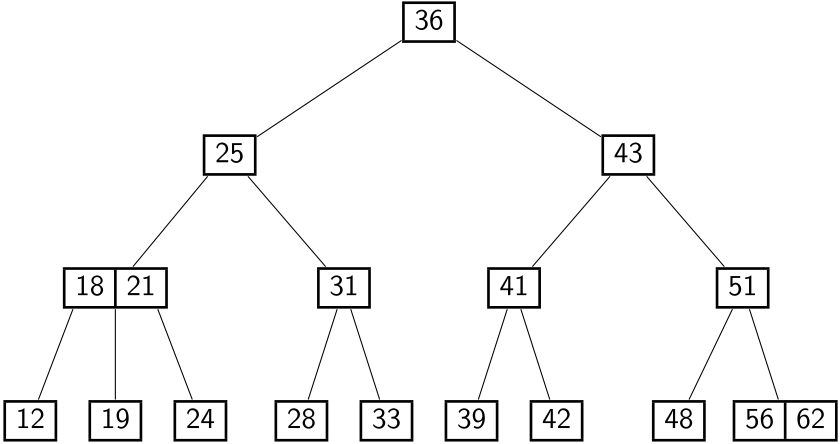


Deletion from a 2-3 Tree

- As with BSTs and AVL trees, we first swap the KVP k with its successor, so that it is now at a leaf ℓ.
- Delete k and one NIL-child from ℓ.
- If ℓ now has 0 keys (*underflow*)
  - If ℓ is the root, simply delete it. Else let p be the parent of ℓ.
  - If some *immediate* sibling u is a 2-node, perform a *transfer*:
    - Find the key k_p in p that is between keys of ℓ and u.
    - "Rotate:" move k_p into ℓ, move adjacent KVP from u into p, and re-arrange children suitably.
  - Otherwise, we *merge* ℓ and a 1-node sibling u:
    - Find the key k_p in parent p between keys of ℓ and u.
    - Combine ℓ and u into one node and move k_p into it.
    - Recurse in p if it now has underflow.

2-3 Tree Deletion

Example:



Outline

1. External Memory
   - Motivation
   - External sorting
   - External Dictionaries
   - 2-3 Trees
   - (a, b)-Trees
   - B-Trees
   - Extendible Hashing

(a, b)-Trees

The 2-3 Tree is a specific type of (a, b)-tree:

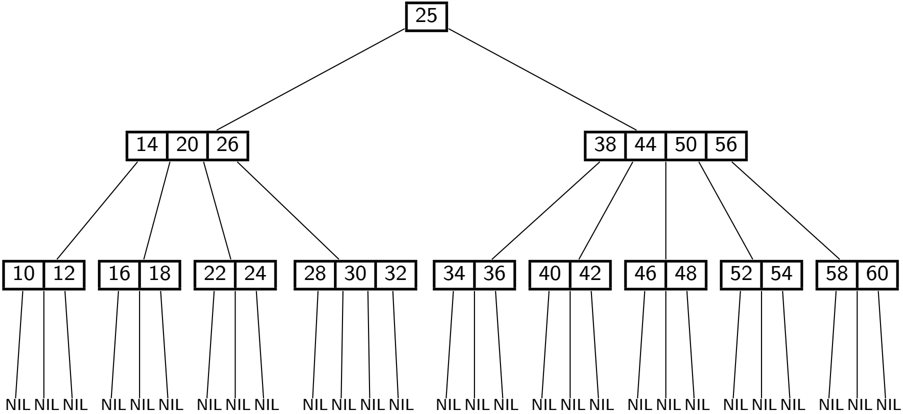
An *(a, b)-tree* satisfies:

- Each internal node has at least a children, unless it is the root. The root has at least 2 children.
- Each internal node has at most b children.
- If a node has k children, then it stores $k - 1$ key-value pairs (KVPs).
- External nodes store no keys and are at the same level.
- The keys in the node are between the keys in the corresponding children.

If $a \geq b/2$, then *search*, *insert*, *delete* work just like for 2-3 trees, after re-defining underflow/overflow to consider the above constraints.

(a, b)-tree example

A $(3, 5)$-tree (it is also a valid $(3, 6)$-tree):



Height of an (a, b)-tree

What is the least number of KVPs in an (a, b)-tree of height-h?
(Height $=$ # levels **not** counting the NIL-level -1)

Level	**Nodes $\geq$**	**Links/node $\geq$**	**KVP/node $\geq$**	**KVPs on level $\geq$**
0	1	2	1	1
1	2	a	$a-1$	$2(a-1)$
2	$2a$	a	$a-1$	$2a(a-1)$
3	$2a^2$	a	$a-1$	$2a^2(a-1)$
$\cdots$	$\cdots$	$\cdots$	$\cdots$	$\cdots$
h	$2a^{h-1}$	a	$a-1$	$2a^{h-1}(a-1)$

$$\text{Total: } n \geq 1 + 2(a-1)\sum_{i=0}^{h-1} a^i = 2a^h - 1$$

Therefore height of tree with n KVPs is $\Theta(\log_a(n)) = \Theta\big(\log n / \log a\big)$.

Outline

1. External Memory
   - Motivation
   - External sorting
   - External Dictionaries
   - 2-3 Trees
   - (a, b)-Trees
   - B-Trees
   - Extendible Hashing

B-trees

A *B-tree of order m* is a $(\lceil m/2 \rceil, m)$-tree.
A 2-3 tree is a B-tree of order 3.

Sedgewick uses M rather than m, but this is confusing since we set M to be the space in main memory.

Analysis (if entire B-tree is stored in main memory):

- Assume each node stores its KVPs and child-pointers in a dictionary that supports $O(\log m)$ search, insert, and delete.
- *search*, *insert*, and *delete* each require $\Theta(\textit{height})$ node operations.
- Height is $O(\log n / \log m)$.
- Each node operation can be done in $O(\log m)$ time.

Total cost is $O\left(\frac{\log n}{\log m} \cdot (\log m)\right) = O(\log n)$.
This is no better than 2-3-trees or AVL-trees.

Dictionaries in external memory

Main applications of B-trees: Store dictionaries in external memory.

Recall: In an AVL tree or 2-3 tree, $\Theta(\log n)$ pages are loaded in the worst case.

Instead, use a B-tree of order m, where m is chosen so that an m-node fits into a single page.

Each operation can be done with $\Theta(\text{height})$ page loads.

The height of a B-tree is $\Theta\big(\log n / \log m\big)$.

This results in *huge* savings of page loads.

B-tree variations

Can reduce page loads even further with two strategies:

- *insert* and *delete* with **pre-emptive splitting/merging**:
  While searching for key k, split/join two nodes that are close to overflow/underflow.
  Then inserting/deleting k will not lead to overflow/underflow.
  $\rightsquigarrow$ no need to recurse back up in the tree, saving those page loads.

- **B$^+$-trees**: Only leaves have KVPs, interior nodes have only keys.
  This means twice as many keys, but we can use a larger m since interior nodes do not hold values.
  We also link the leaves sequentially.

Also of note: **Red-black trees**. These are (2,4)-trees where 2-nodes and 3-nodes are replaced by 2 or 3 binary nodes, and colours are used to indicate the types of nodes. These are balanced binary search trees that are faster than AVL-trees in practice.

Outline

Hashing in External Memory

As before, if we have a *very large* dictionary that must be stored externally, how can we hash and minimize disk transfers?

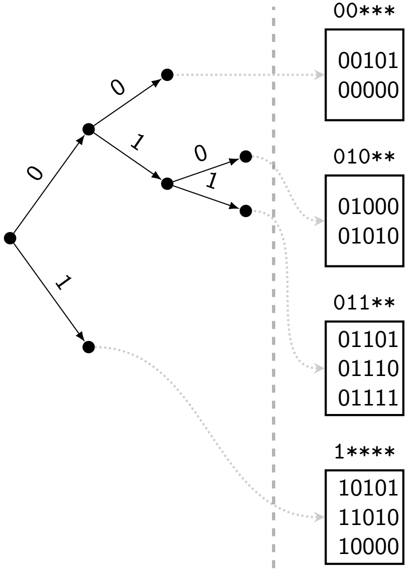
Most hash strategies access many pages (data is scattered).

Exception: **Linear Probing**.

- All hash table accesses will usually be in the same page.
- But α must be kept small to avoid clustering, so there is a lot of wasted space.
- And re-hashing must load *all* pages.

New Idea: **Extendible Hashing**. Key idea: store trie of hash-values to link to correct page.

External hashing with tries – Overview



Assumption: Hash-function has values in $\{0, 1, \ldots, 2^L - 1\}$.

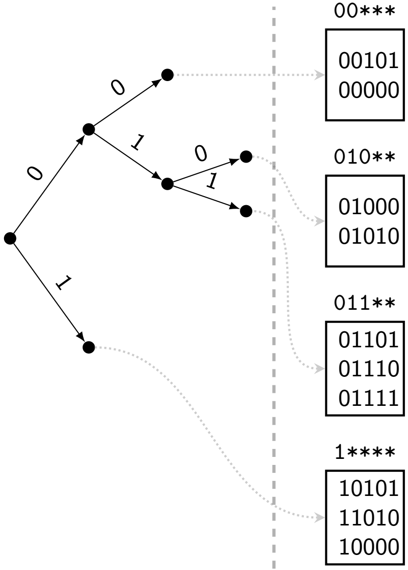
Interpret all hash-values as bitstrings of length L.

Build trie D (the *directory*) of hash-values in internal memory.

Stop splitting in trie whenever at most m items are left, where m is the maximum number of items that fit in one page.

Each leaf of D refers to *page* in external memory that stores the items.

External hashing with tries – Details

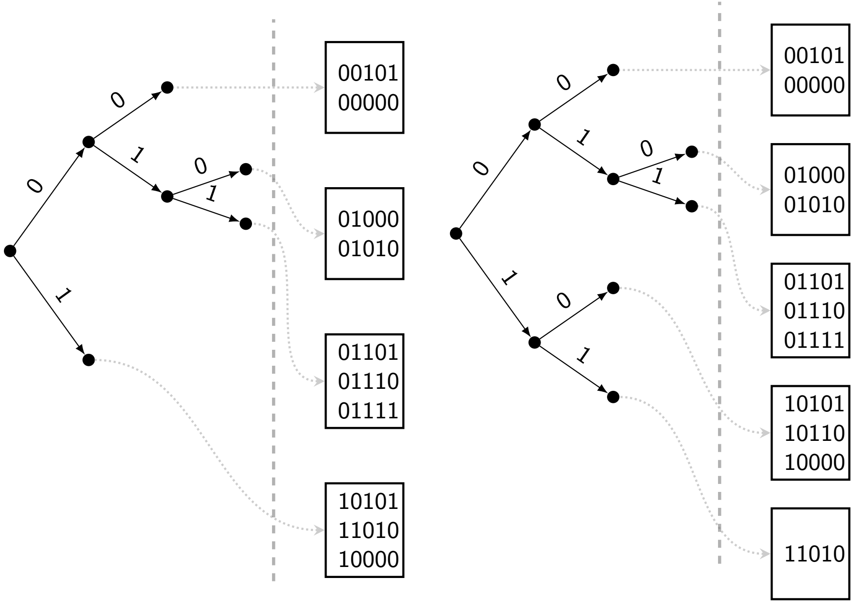


Search(x): Compute $h(x)$. Search for $h(x)$ in D until we reach leaf ℓ. Load page at ℓ and search in it.
1 page load.

Insert(x): Search for x and load page, then insert x. If this exceeds page-capacity, split at trie-node and split pages (possibly repeatedly).
Typically $1 - 2$ **page loads**.

Delete(x): Search for x and load page, then mark deleted (lazy deletion). Optional: combine underfull pages.
1 page load.

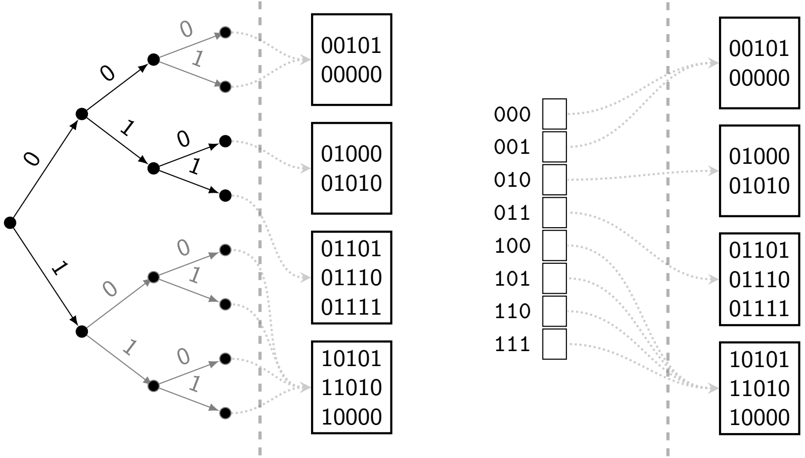
Insert(10110)



Extendible hashing: saving space

We can save links (hence space) in main memory with two tricks:

- Expand the trie so that all leaves have the same depth (*order* d). Multiple leaves may point to same page.
- Store *only* the leaves, and in an array D.



Summary of extendible hashing

- Directory is much smaller than total number of stored keys
  $\rightarrow$ should fit in main memory.
  (If it does not, then one could use a B-tree for the dictionary.)
- Only 1 or 2 page transfers for *any* operation.
- To make more space, we only add one block.
  Rarely change the size of the directory.
  Never have to move all items. (in contrast to re-hashing!)
- Space usage is not too inefficient: one can show that under uniform hashing each block is expected to be 69% full.