CS 240 – Data Structures and Data Management

Module 11: External Memory

T. Biedl  M. Petrick  O. Veksler
Based on lecture notes by many previous cs240 instructors

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

Winter 2019

References: Goodrich & Tamassia 20.1-20.3, Sedgewick 16.4
Outline

1. External Memory
   - Motivation
   - External sorting
   - External Dictionaries
   - 2-4 Trees
   - $a$-$b$-Trees
   - B-Trees
   - Extendible Hashing
Outline

1. External Memory
   - Motivation
     - External sorting
     - External Dictionaries
     - 2-4 Trees
     - $a$-$b$-Trees
     - B-Trees
     - Extendible Hashing
Different levels of memory

Current architectures:
- registers (very fast, very small)
- cache L1, L2 (still fast, less small)
- main 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 *block transfers* (“probes”, “disk transfers”, “page loads”)
The External-Memory Model (EMM)

Cost of computation:
\[ \# \text{“block transfers”} = \# \text{blocks transferred between internal and external memory} \]

- External memory – size unbounded
- Internal memory – size \( M \)
- CPU
  - Fast random access
  - Slow access only in \textit{blocks} of \( B \) cells

\[ \text{Cost of computation: } \# \text{“block transfers”} = \# \text{blocks transferred between internal and external memory} \]
Outline

1 External Memory
   - Motivation
   - External sorting
     - External Dictionaries
   - 2-4 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
  $\Rightarrow$ typically one block transfer 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

\[
d\text{-Way-Merge}(S_1, \ldots, S_d)
\]
\[
S_1, \ldots, S_d \text{ are sorted sets (arrays/lists/stacks/queues)}
\]
\[
1. \quad P \leftarrow \text{empty min-priority queue}
\]
\[
2. \quad S \leftarrow \text{empty set}
\]
\[
3. \quad \text{for } i \leftarrow 1 \text{ to } d \text{ do}
\]
\[
4. \quad \quad P.\text{insert}((\text{first element of } S_i, i))
\]
\[
5. \quad \quad \text{while } P \text{ is not empty do}
\]
\[
6. \quad \quad \quad (x, i) \leftarrow \text{deleteMin}(P)
\]
\[
7. \quad \quad \quad \text{remove } x \text{ from } S_i \text{ and append it to } S
\]
\[
8. \quad \quad \quad \text{if } S_i \text{ is not empty do}
\]
\[
9. \quad \quad \quad \quad P.\text{insert}((\text{first element of } S_i, i))
\]

- Standard merge (within mergesort) uses \(d = 2\)
- \(d > 2\) could be used in internal memory as well, but the extra time for \textit{deleteMin} means the overall run-time is no better.
Mergesort in external memory

External \((B = 2)\):

\[
\begin{array}{cccccccccccccccc}
\end{array}
\]

Internal \((M = 8)\):

\[
\begin{array}{cccccccc}
\end{array}
\]

1. Create \(n/M\) sorted runs of length \(M\). \(\Theta(n/B)\) block transfers
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 \(d\) → one round of merging. \(\Theta(n/B)\) block transfers
4. \(\log_d(n/M)\) rounds of merging create sorted array.
Mergesort with external memory

Total \# block transfers: \( O(\log_d(n) \cdot n/B) \).

Assuming the EMM, one can prove lower bounds!

- \( \Omega\left(\frac{n}{B}\right) \) block transfers required to scan \( n \) elements.
- \( \Omega\left(\frac{n}{B} \log_{M/B}\left(\frac{n}{B}\right)\right) \) block transfers required for comparison-based sort
  - We don’t prove that here.

- \( d \)-way Mergesort with \( d \approx M/B \) is optimal (up to constant factors)!
Outline

1. External Memory
   - Motivation
   - External sorting
   - External Dictionaries
     - 2-4 Trees
     - $a$-$b$-Trees
     - B-Trees
     - Extendible Hashing
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, \( \Theta(\log n) \) blocks are loaded in the worst case.
- Better solution: do more in single node \( \rightsquigarrow \) B-trees

- First consider special case of B-trees: 2-4 trees
  - 2-4-trees would also be interesting for implementing ADT Dictionaries in internal memory (may be even faster than AVL-trees)
  - We first analyze their performance in internal memory, and then (for B-trees) in external memory.
Outline

1 External Memory
   • Motivation
   • External sorting
   • External Dictionaries
   • 2-4 Trees
     • a-b-Trees
     • B-Trees
     • Extendible Hashing
2-4 Trees

A 2-4 Tree is a balanced search tree that is not necessarily binary.

**Structural properties:**
- Every node is either
  - 1-node: one KVP and two subtrees (possibly empty), or
  - 2-node: two KVPs and three subtrees (possibly empty), or
  - 3-node: three KVPs and four subtrees (possibly empty).
- All empty subtrees are at the same level.

Height-balance strictly enforced, but allow 3 types of nodes!

**Order property:** The keys at a node are between the keys in the subtrees.
2-4 Tree operations

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

\[
\text{24TreeSearch}(k, v \leftarrow \text{root}, p \leftarrow \text{empty subtree})
\]

\[
k: \text{key to search}, v: \text{node where we search}, p: \text{parent of } v
\]

1. if \( v \) represents empty subtree
2. return “not found, would be in \( p \)”
3. Let \( T_0, k_1, \ldots, k_d, T_d \) be keys and subtrees at \( v \), in order
4. if \( k \geq k_1 \)
5. \( i \leftarrow \text{maximal index such that } k_i \leq k \)
6. if \( k_i = k \)
7. return “at \( i \)th key in \( v \)”
8. else \( 24\text{TreeSearch}(k, T_i, v) \)
9. else \( 24\text{TreeSearch}(k, T_0, v) \)
2-4 Tree operations

24TreeInsert(\(k\))

1. \(v \leftarrow 24TreeSearch(\(k\))\) // leaf where \(k\) should be
2. Add \(k\) and an empty subtree in key-subtree-list of \(v\)
3. while \(v\) has 4 keys (overflow → node split)
4. Let \(T_0, k_1, \ldots, k_4, T_4\) be keys and subtrees at \(v\), in order
5. if (\(v\) has no parent) create an empty parent of \(v\)
6. \(p \leftarrow \) parent of \(v\)
7. \(v' \leftarrow \) new node with keys \(k_1, k_2\) and subtrees \(T_0, T_1, T_2\)
8. \(v'' \leftarrow \) new node with key \(k_4\) and subtrees \(T_3, T_4\)
9. Replace \(\langle v \rangle\) by \(\langle v', k_3, v'' \rangle\) in key-subtree-list of \(p\)
10. \(v \leftarrow p\)
Example: Insertion in a 2-4 tree

**Example:** \texttt{24TreeInsert}(17)
Deletion from a 2-4 Tree

$24TreeDelete(k)$

1. $w \leftarrow 24TreeSearch(k)$  // node containing $k$
2. $v \leftarrow$ leaf containing predecessor or successor $k'$ of $k$
3. Replace $k$ by $k'$ in $w$, delete $k'$ and empty subtree in $v$
4. while $v$ has 0 keys (underflow)
   5. if $v$ is the root, delete it and break
   6. $p \leftarrow$ parent of $v$
   7. if $v$ has a sibling $u$ with 2 or more keys (transfer/rotate)
      8. if $u$ is right sibling
         Replace key $k$ in $p$ by $u.k_1$
         Remove $\langle u.T_0, u.k_1 \rangle$ from $u$, append $\langle k, u.T_0 \rangle$ to $v$
      11. else ...  // symmetrically with left sibling
   12. else (merge & repeat)
      13. if $v$ has right sibling $u$
         $v' \leftarrow$ new node with list $\langle v.T_0, k, u.T_0, u.k_1, u.T_1 \rangle$
         replace $\langle v, k, u \rangle$ by $\langle v \rangle$ in $p$
      16. $v \leftarrow p$
      17. else ...  // symmetrically with left sibling
2-4 Tree Deletion

Example:

```
36
/  \
25 /  \
18 21 /  31 /  41 /  51
12 19 24 28 33 39 42 48 56 62
```

Biedl, Petrick, Veksler (SCS, UW)
Outline

1 External Memory
   • Motivation
   • External sorting
   • External Dictionaries
   • 2-4 Trees
   • a-b-Trees
   • B-Trees
   • Extendible Hashing
**a-b-Trees**

The 2-4 Tree is an \( a-b \)-tree for \( a = 2 \) and \( b = 4 \).

An \( a-b \)-tree satisfies:

- Each node has at least \( a \) subtrees, unless it is the root. The root has at least 2 subtrees.
- Each node has at most \( b \) subtrees.
- If a node has \( k \) subtrees, then it stores \( k - 1 \) key-value pairs (KVPs).
- Empty subtrees are at the same level.
- The keys in the node are between the keys in the corresponding subtrees.

If \( a \leq \lceil b/2 \rceil \), then search, insert, delete work just like for 2-4 trees, after re-defining underflow/overflow to consider the above constraints.
a-b-tree example

A 3-6-tree
a-b-tree insertion

Insert(55):
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 empty subtrees)

<table>
<thead>
<tr>
<th>Level</th>
<th>Nodes $\geq$</th>
<th>Links/node $\geq$</th>
<th>KVP/node $\geq$</th>
<th>KVPs on level $\geq$</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>2</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>2</td>
<td>$a$</td>
<td>$a - 1$</td>
<td>$2(a - 1)$</td>
</tr>
<tr>
<td>2</td>
<td>$2a$</td>
<td>$a$</td>
<td>$a - 1$</td>
<td>$2a(a - 1)$</td>
</tr>
<tr>
<td>3</td>
<td>$2a^2$</td>
<td>$a$</td>
<td>$a - 1$</td>
<td>$2a^2(a - 1)$</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>$h$</td>
<td>$2a^{h-1}$</td>
<td>$a$</td>
<td>$a - 1$</td>
<td>$2a^{h-1}(a - 1)$</td>
</tr>
</tbody>
</table>

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 $\Theta(\log_a(n)) = \Theta(\log n / \log a)$. 

Biedl, Petrick, Veksler (SCS, UW)
Outline

1. External Memory
   - Motivation
   - External sorting
   - External Dictionaries
   - 2-4 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 2-4 tree is a B-tree of order 4.

**Analysis** (if entire B-tree is stored in internal memory):
- Assume each node stores its KVPs and subtree-pointers in a dictionary that supports $O(\log m)$ search, insert, and delete.
- *search, insert, and delete* each require $\Theta(\text{height})$ node operations.
- Height is $O(\log n/ \log(m/2)) = O(\log n/\log m)$.
- 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 2-4-trees or AVL-trees.
Dictionaries in external memory

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

**Recall:** In an AVL tree or 2-4 tree, $\Theta(\log n)$ blocks 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 block. Typically $m \in \Theta(B)$.

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

The height of a B-tree is $\Theta(\log_m n) = \Theta(\log_B n)$.

This results in *huge* savings of block transfers.
B-tree variations

For practical purposes, some variations are better:

- **B-trees with pre-emptive splitting/merging:**
  - During search for insert, split any node close to overflow.
  - During search for delete, merge any node close to underflow.
  
  → can insert/delete at leaf and stop, this halves block transfers.

- **$B^+$-trees:** Only leaves have KVPs, link leaves sequentially.
  - Interior nodes store duplicates of keys to guide search-path.
  - Up to twice as many keys stored
  
  → larger $m$ since interior nodes do not hold values.

- **Cache-oblivious** trees: What if we do not know $B$?
  Build a hierarchy of binary trees
  - Each node $v$ in binary tree $T$ “hides” a binary tree $T'$ of size $\Theta(\sqrt{n})$
  
  → achieves $O(\log_B(n))$ block transfers *without* knowing $B$. 
Outline

1 External Memory
   • Motivation
   • External sorting
   • External Dictionaries
   • 2-4 Trees
   • $a$-$b$-Trees
   • B-Trees
   • Extendible Hashing
Dictionaries for Integers in External Memory

- Recall: Direct Addressing allowed for $O(1)$ insert and delete if keys are integers in \{0, \ldots, U − 1\}
- If keys are too big, hashing was used to map keys to (smaller) integers.
- This does not adapt well to external memory.
  - Most hash strategies access many blocks (probe sequence is scattered)
  - Even those that do not (Linear Probing, Cuckoo hashing) need to re-hash to keep $\alpha$ small.
  - And re-hashing must load all blocks.
- New Idea: Store trie of links to blocks of integers.

(This is also called **extendible hashing**, because its primary use is for dictionaries that store integers that result from hashing.)
Tries of blocks – Overview

Assumption: We store integers in \( \{0, 1, \ldots, 2^L - 1\} \).

Interpret all integers as bitstrings of length \( L \).

Build trie \( D \) (the directory) of integers in internal memory.

Stop splitting in trie when remaining items fit in one block.

Each leaf of \( D \) refers to block in external memory that stores the items.
**External hashing with tries – Details**

**Search**(x): Search for \((x)_2\) in \(D\) until we reach leaf \(\ell\). Load block at \(\ell\) and search in it.

**1 block transfer.**

**Insert**(x): Search for \(x\) and load block, then insert \(x\). If this exceeds block-capacity, split at trie-node and split blocks (possibly repeatedly).

**Typically 1 – 2 block transfers.**

**Delete**(x): Search for \(x\) and load block, then mark deleted (lazy deletion). Optional: combine underfull blocks.

**1 block transfer.**
Insert(10110)
Extendible hashing: Insert

$$\text{Extendible-Hashing-Insert}(k)$$

\(k\): integer in \(\{0, \ldots, 2^L - 1\}\)

1. Convert \(k\) to length-\(L\) bitstring \(w\)
2. \(\ell \leftarrow \text{Trie-Search}(D, w)\) \hspace{1cm} // leaf where \(w\) would be
3. \(d \leftarrow \text{depth of } \ell \text{ in } D\) \hspace{1cm} // local depth
4. transfer block \(P\) that \(\ell\) refers to
5. **while** \(P\) is full (cannot add item without exceeding block-size)
6. Split \(P\) into two blocks \(P_0\) and \(P_1\) by \((d + 1)\text{st digit}\)
7. Create two children \(\ell_0\) and \(\ell_1\) of \(\ell\), linked to \(P_0\) and \(P_1\)
8. \(d \leftarrow d+1, \ell \leftarrow \ell_w[d], P \leftarrow P_w[d]\)
9. insert \(w\) into \(P\)
Extendible hashing: saving space

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

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

![Diagram showing extendible hashing](image-url)
Summary of extendible hashing

- Directory is much smaller than total number of stored keys → should fit in internal memory.
  (If it does not, then one could use a B-tree for the dictionary.)
- Only 1 or 2 block 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 distribution assumption each block is expected to be 69% full.