Basic External Memory Data Structures

Zorienh Soltani

Yazd University

Fall-1389
2.3 B-trees
2.4 Hashing Based Dictionaries
2.5 Dynamization Techniques
B-trees
Introduction

- We want search trees of **large degree** because of using all the information we get when reading a block to guide the search.
- B-trees are a generalization of balanced binary search trees to balanced trees of degree $\Theta(B)$.
- $N$: the size of the key set and $B$: the number of keys or pointers that fit in one block.

![Diagram of External Search Trees](image-url)
In a B-tree all leaves have the same distance to the root.

Level of a node: its distance to its descendant leaves.

Weight of node $v$: the number of leaves subtree of node $v$, is shown by $w(v)$. 

Introduction (continue)

- In a B-tree all leaves have the same distance to the root
- **Level** of a node: its distance to its descendant leaves
- **Weight** of node $v$: the number of leaves subtree of node $v$, is shown by $w(v)$
In a B-tree all leaves have the same distance to the root.

*Level* of a node: its distance to its descendant leaves.

*Weight of node* $v$: the number of leaves subtree of node $v$, is shown by $w(v)$. 
In a B-tree all leaves have the same distance to the root

*Level* of a node: its distance to its descendant leaves

*Weight of node* \( v \): the number of leaves subtree of node \( v \), is shown by \( w(v) \)
B-trees

Definition

T is a weight-balanced B-tree with branching parameter \( b \) and leaf parameter \( k \), \( (b \geq 4 \text{ and } k \geq 0) \) if:

- All leaves of \( T \) have the same depth and weight between \( k \) and \( 2k - 1 \)
- An internal node on level \( l \) has weight less than \( 2b^l k \)
- An internal node on level \( l \) except for the root has weight greater than \( \frac{1}{2} b^l k \)
- The root has more than one child
B-trees

Limitation on weight results Limitation on degree of each node

Degree of each node is between \( b \) and \( 4b \)

The degree of any non-root node is \( \Theta(b) \)
Limitation on weight results in a limitation on the degree of each node. The degree of each node is bounded by $b$ and $4b$. The degree of any non-root node is $\Theta(b)$. The formula $k < w(f) < 2^k - 1$ represents the weight condition for each level $f$.
Limitation on weight results in

\[ \frac{1}{2} b^k < w(v) < 2b^k \]

Limitation on degree of each node

\[ k < w(f) < 2k - 1 \]
- Limitation on weight results in limitation on degree of each node.
- Degree of each node is between $\frac{b}{4}$ and $4b$.
- Limitation on weight results in limitation on degree of each node.
- Degree of each node is between $\frac{b}{4}$ and $4b$.
- The degree of any non-root node is $\Theta(b)$.
The New B-tree is introduced by our book

The result branching parameter is: $b = B$

And we assume leaf parameter: $k = 2$
The New B-tree is introduced by our book

The result branching parameter is: \( b = \frac{B}{8} \)

And we assume leaf parameter: \( k = 2 \)
The New B-tree is introduced by our book (continue)

- An internal node on level \(i\) has weight less than \(4\left(\frac{B}{8}\right)^i\)
- An internal node on level \(i\) except for the root has weight greater than \(\left(\frac{B}{8}\right)^i\)
- Any node has less than \(B/2\) children
- Any non-root node has greater than \(B/32\) children
Searching a B-tree

- In a node $v$ stores sorting keys $k_1, ..., k_{d_v - 1}$
- The $i$th subtree of $v$ stores keys $k$ with $k_{i-1} \leq k < k_i$ (defining $k_0 = -\infty$ and $k_{d_v} = \infty$).
- The information in a node suffices to determine in which subtree to continue a search.
- The worst-case number of I/Os needed for searching a B-tree equals the worst-case height of a B-tree, at most $1 + \lceil \log_b N \rceil$. 
"report all keys in the range \([a,b]\)"

- **Search for the key** \(a\), which will lead to the smallest key \(x \geq a\)
- **Traverse the linked list** starting with \(x\) and report all keys smaller than \(b\)
- **of I/Os** of Rang queries (output sensitivity): \(O(\log_b N + Z/B)\)
"report all keys in the range \([a,b]\)"

- Search for the key \(a\), which will lead to the smallest key \(x \geq a\)
- Traverse the linked list starting with \(x\) and report all keys smaller than \(b\)
- of I/Os of Rang queries (output sensitivity): \(O(\log_b N + Z/B)\)
"report all keys in the range \([a,b]\)"

- Search for the key \(a\), which will lead to the smallest key \(x \geq a\)
- Traverse the linked list starting with \(x\) and report all keys smaller than \(b\)
- \(O(\log_b N + Z/B)\) of I/Os of Rang queries (output sensitivity)
"report all keys in the range [a, b]"

- Search for the key \( a \), which will lead to the smallest key \( x \geq a \)
- Traverse the linked list starting with \( x \) and report all keys smaller than \( b \)
- of I/Os of Rang queries (output sensitivity): \( O(\log_b N + Z/B) \)
Range Reporting (continue)

Two Notes

1. Optimal solution is based on hashing data structures that performs in $O(1 + Z/B)$.

2. Optimal output sensitivity fails when query changes to "report the first $Z$ keys in the range $[a,b]$"
Inserting and Deleting Keys in a B-tree

Inserting Key $x$

- Search for the key $x$, find node $v$ that is parent of $x$
- Insert the key $x$ to node $v$
- If at level $i$, $w(v) = 2b^l k$ (overweight), we rebalance it by "split"
- We split a node $v$ to two new nodes $u, u'$
- starting from the bottom and going up
Inserting and Deleting Keys in a B-tree

**Inserting Key x**
- Search for the key x, find node v that is parent of x
- **Insert the key x to node v**
- If at level i, \(w(v) = 2^l k\) (overweight), we rebalance it by "split"
- We split a node v to two new nodes u, u'
- starting from the bottom and going up
Inserting and Deleting Keys in a B-tree

Inserting Key x

- Search for the key x, find node v that is parent of x
- Insert the key x to node v
- If at level l, \( w(v) = 2b^l k \) (overweight), we rebalance it by "split"
- We split a node v to two new nodes u, u'
- starting from the bottom and going up
Inserting and Deleting Keys in a B-tree

**Inserting Key x**
- Search for the key x, find node v that is parent of x
- Insert the key x to node v
- If at level l, \( w(v) = 2b^l k \) (overweight), we rebalance it by "split"
- We split a node v to two new nodes u, u'
- starting from the bottom and going up
Inserting and Deleting Keys in a B-tree

Inserting Key $x$

- Search for the key $x$, find node $v$ that is parent of $x$
- Insert the key $x$ to node $v$
- If at level $l$, $w(v)=2b^l k$ (overweight), we rebalance it by "split"
- We split a node $v$ to two new nodes $u, u'$
- Starting from the bottom and going up
Inserting and Deleting Keys in a B-tree

Inserting Key $x$

- Search for the key $x$, find node $v$ that is parent of $x$
- Insert the key $x$ to node $v$
- If at level $i$, $w(v) = 2b^l k$ (overweight), we rebalance it by "split"
- We split a node $v$ to two new nodes $u, u'$
- starting from the bottom and going up
Inserting and Deleting Keys in a B-tree

Inserting Key $x$

- Search for the key $x$, find node $v$ that is parent of $x$
- Insert the key $x$ to node $v$
- If at level $i$, $w(v) = 2b^l k$ (overweight), we rebalance it by "split"
- We split a node $v$ to two new nodes $u, u'$
- starting from the bottom and going up
Inserting key $x$ (continue)

- $b^l k - 2b^{l-1} k \langle w(u), w(u') \langle b^l k + 2b^{l-1} k$
- Since $b \geq 4$
- $\frac{1}{2} b^l k \langle w(u), w(u') \langle \frac{3}{2} b^l k$
- The weight of each of these new nodes $(u, u')$ is $\Omega(b^l)$
Inserting and Deleting Keys in a B-tree (continue)

<table>
<thead>
<tr>
<th>Deleting Key x (fuse)</th>
</tr>
</thead>
<tbody>
<tr>
<td>- Search for the key x to find the internal node v that is parent x</td>
</tr>
<tr>
<td>- Delete the key x from node v</td>
</tr>
<tr>
<td>- If at level l, ( w(v) = \frac{1}{2} b^l k ) (underweight), we will rebalance it by ”fuse” or ”share” operations</td>
</tr>
<tr>
<td>- starting from the bottom and going up</td>
</tr>
<tr>
<td>- Node w: one of its nearest sibling of node v</td>
</tr>
<tr>
<td>- If ( w(w) \leq \frac{5}{4} b^l k ) we do ”fuse” operation</td>
</tr>
</tbody>
</table>
Deleting Keys in a B-tree (fuse)
Deleting Keys in a B-tree (fuse)

An underweight node

$\frac{1}{2} b^{l+1} k \ldots 2b^{l+1} k$

$\frac{1}{2} b^l k$

Zorieh Soltani (Yazd University)
Basic External Memory Data Structures
Fall-1389 16 / 50
Deleting Keys in a B-tree (fuse)
Deleting Keys in a B-tree (fuse)

Fuse two nodes $\frac{1}{2} b^{l+1} k \ldots 2b^{l+1} k$

Fuse two nodes $b^l k \ldots \frac{7}{4} b^l k$
Deleting Keys in a B-tree (share)

- If $\frac{5}{4}b^lk \leq w(w) \leq 2b^lk$ we do "share" operation.
- We have two new nodes $u, u'$ as the result of "share".
- $w(u) = \frac{7}{8}b^lk - 2b^{l-1}k$
- $w(u') = \frac{5}{4}b^lk + 2b^{l-1}k$
- The weight of each of them ($u, u'$) is $\Omega(b^l)$.
Deleting Keys in a B-tree (share)

- If \( \frac{5}{4} b^l k \langle w(w) \langle 2b^l k \) we do "share" operation
- We have two new nodes \( u, u' \)
- Result of "share"
- \( w(u) = \frac{7}{8} b^l k - 2b^{l-1} k \)
- \( w(u') = \frac{5}{4} b^l k + 2b^{l-1} k \)
- The weight of each of them \( (u, u') \) is \( \Omega(b^l) \)
Deleting Keys in a B-tree (share)

- If \( \frac{5}{4} b^l k \langle w(w) \langle 2 b^l k \) we do "share" operation
- We have two new nodes \( u, u' \) result of "share"
- \( w(u) = \frac{7}{8} b^l k - 2 b^{l-1} k \)
- \( w(u') = \frac{5}{4} b^l k + 2 b^{l-1} k \)
- The weight of each of them \( (u, u') \) is \( \Omega(b^l) \)
Deleting Keys in a B-tree (share)

- If $\frac{5}{4} b^l k < w(w) < 2 b^l k$ we do "share" operation
- We have two new nodes $u, u'$ result of "share"
- $w(u) = \frac{7}{8} b^l k - 2 b^{l-1} k$
- $w(u') = \frac{5}{4} b^l k + 2 b^{l-1} k$
- The weight of each of them $(u, u')$ is $\Omega(b^l)$
Deleting Keys in a B-tree (share)

- If $\frac{5}{4} b^l k < w(w) \langle 2b^l k$ we do "share" operation
- We have two new nodes $u, u'$ result of "share"
- $w(u) = \frac{7}{8} b^l k - 2b^{l-1} k$
- $w(u') = \frac{5}{4} b^l k + 2b^{l-1} k$
- The weight of each of them ($u, u'$) is $\Omega(b^l)$
Analysis of inserting and deleting in B-tree

- The cost of rebalancing a node: $O(1)$ I/Os
- The total cost of B-tree rebalancing: $O(\log_b N)$ I/Os
- We have in fact shown something stronger
- The weight of node $v$ at level $i$, $W = \Theta(b^i)$
- To assume $S$: an auxiliary data structure used when searching in the $v$’s subtree
- When $v$ is rebalanced we spend $f(W)$ I/Os to compute $S$
The rebalancing operation have $\Omega(W)$ insertions and deletions in $v$’s subtree and also in $S$.

The amortized cost of maintaining $S: O(f(W)/W)$ I/Os per node on the search path of an update.

or $O\left(\frac{f(W)}{W} \log^N_b\right)$ I/Os per update.

As an example, if $f(W)=O(W/B)$ I/Os.

The amortized cost per update is $O\left(\frac{1}{B} \log^N_b\right)$ I/Os.

that this is negligible.
B-tree Variants

1. Parent Pointers and Level Links
   - Maintain a pointer to the parent of each node
   - Maintain all nodes at each level with a doubly linked list
   - One application of these pointers is a "finger search"
     - Given a leaf v in the B-tree, search for another leaf w
     - Q: the number of leaves between v and w
   - The number of I/Os:
     \[ O(\log_b Q) \]

2. String B-trees
   - We have assumed that the B-tree's keys have fixed length
   - In some applications the keys are strings of unbounded length
   - all the usual B-tree operations can be efficiently supported in this setting
B-tree Variants

1. Parent Pointers and Level Links

- Maintain a pointer to the parent of each node
- Maintain all nodes at each level with a doubly linked list
- One application of these pointers is a "finger search"
- Given a leaf v in the B-tree, search for another leaf w
- Q: the number of leaves between v and w
- The number of I/Os: $O(\log_b^Q)$
B-tree Variants

1. Parent Pointers and Level Links
- Maintain a pointer to the parent of each node
- Maintain all nodes at each level with a doubly linked list
- One application of these pointers is a "finger search"
- Given a leaf v in the B-tree, search for another leaf w
- Q: the number of leaves between v and w
- The number of I/Os: $O(\log_b Q)$

2. String B-trees
- We have assumed that the B-tree’s keys have fixed length
- In some applications the keys are strings of unbounded length
- all the usual B-tree operations, can be efficiently supported in this setting
3. Divide and Merge Operations

- We have two useful applications
- Divide a B-tree into two parts
- Merge two B-trees "glue"
- These operations can be supported in $O(\log_b^N)$ I/Os
Batched Dynamic Problems

- B-trees answer queries in an on-line fashion
- In batched dynamic problems a *batch* of updates and queries is provided to the data structure
- Only at the end of the batch, the data structure delivers the answers

**The batched range searching**
- Given a sequence of insertions and deletions of integers
- Each query of integers is compared with the sequence and reported
Buffer trees

- The buffer tree technique has been used for I/O optimal algorithms
- Each internal node has an buffer with size $\Theta(M)$
- A buffer tree has degree $\Theta(M/B)$
- Leaves contain $\Theta(B)$ keys
- Root buffer reside entirely on main memory
- Non-root buffers reside entirely on external memory
How does a buffer tree work?

```
O(\log_{M/B} N/B) = \Theta(B)
```

```
root
main memory
```

```
\Theta(M/B)
```

```
O(B)
```

```
``
How does a buffer tree work?

\[ O(\log \frac{M}{B}) \]

\[ \Theta(M/B) \]

\[ \Theta(B) \]

main memory

\[ M \text{ elements} \]
How does a buffer tree work?

\[ O(\log \frac{M}{B}) \]

\[ \Theta(B) \]

\[ \Theta(\frac{M}{B}) \]
How does a buffer tree work?

The buffer gets full
It is flushed

\[ O(\log_{M/B} N) \]

\[ \Theta(M/B) \]

\[ \Theta(B) \]

main memory

\( M \) elements

\( \log_{M/B} \)

\( N \)

\( B \)
How does a buffer tree work?

\[ O(\log \frac{M}{B}) \quad N_B \]

\[ \Theta(M/B) \]

\[ \Theta(B) \]

main memory

\[ \frac{M}{B} \]
How does a buffer tree work?

\[ \Theta(B) \]

\[ \Theta(M/B) \]

main memory

\[ O(\log M/B) \]

\[ N/B \]
How does a buffer tree work?

\[ \Theta(B) \]

\[ \Theta(M/B) \]

\[ O(\log \frac{M}{B}) \]

main memory

\[ \square \text{ M elements} \]
How does a buffer tree work?
How does a buffer tree work?

$O(\log \frac{M}{B})$  $\frac{N}{B}$

$\Theta(B)$

$\Theta(\frac{M}{B})$

main memory

$M$ elements
How does a buffer tree work?

\[ \Theta(B) \]

\[ \Theta(M/B) \]

\[ O(\log_{M/B} N/B) \]

main memory

\( M \) elements
How does a buffer tree work?

\[ \Theta(B) \]

\[ \Theta\left(\frac{M}{B}\right) \]

main memory

\[ O(\log \frac{M}{B}) \]

Zorieh Soltani (Yazd University)
How does a buffer tree work?

If there are too few or too many children, rebalancing operations are performed.

\[ O(\log \frac{M}{B}) \]

\[ \Theta(M/B) \]

main memory

\[ \Theta(B) \]

If there are too few or too many children, rebalancing operations are performed.
I/O Analysis for Buffer tree

The cost of flushing a buffer is $O(M/B)$ I/Os for reading the buffer and $O(M/B)$ I/Os for writing the operations to the buffers of the children. The cost of all flushes is $O(1/B) \log(N/M) I/Os$ per operation.

A flushing costs $O(1/B)$ I/Os per operation in the buffer. The total cost of rebalancing during $N$ updates is $O(N/B)$ I/Os. The cost of a rebalancing operation on a node is $O(M/B)$ I/Os. The number of nodes that need rebalancing operations during $N$ updates is $O(N/M)$. 

---

Zorieh Soltani (Yazd University)  
Basic External Memory Data Structures  
Fall-1389  
25 / 50
I/O Analysis for Buffer tree

The cost of flushing a buffer

- $O(M/B)$ I/Os for reading the buffer
- $O(M/B)$ I/Os for writing the operations to the buffers of the children

The total cost of rebalancing during $N$ updates is $O(N/B)$ I/Os

The cost of a rebalancing operation on a node is $O(M/B)$ I/Os

Number of nodes that need to rebalancing operations during $N$ updates is $O(N/M)$
I/O Analysis for Buffer tree

The cost of flushing a buffer
- \( O(M/B) \) I/Os for reading the buffer
- \( O(M/B) \) I/Os for writing the operations to the buffers of the children

The cost of all of flushes \( O\left(\frac{1}{B} \log_{\frac{N}{MB}}\right) \) I/Os per operation
- A flushing costs \( O(1/B) \) I/Os per operation in the buffer
I/O Analysis for Buffer tree

The cost of flushing a buffer
- $O(M/B)$ I/Os for reading the buffer
- $O(M/B)$ I/Os for writing the operations to the buffers of the children

The cost of all of flushes $O\left(\frac{1}{B} \log_{\frac{M}{B}} \frac{N}{B} \right)$ I/Os per operation
- A flushing costs $O(1/B)$ I/Os per operation in the buffer

The total cost of rebalancing during $N$ updates is $O(N/B)$ I/Os
- The cost of a rebalancing operation on a node is $O(M/B)$ I/Os
- Number of nodes that need to rebalancing operations during $N$ updates is $O(N/M)$
Priority Queues

- The basic operations insertion of a key, finding the smallest key, and deleting the smallest key
- Sometimes additional operations are supported, such as deleting an arbitrary key and decreasing the value of a key
- We use buffering technique for priority queue
- The entire buffer of the root node and the $O(M/B)$ leftmost leaves are always kept in internal memory
How does priority queue using buffer tree work?

- All buffers on the path from the root to the leftmost leaf must be empty.
- For this, whenever the root is flushed we also flush all buffers down the leftmost path.

\[ \Theta(B) \] 
\[ \Theta(M/B) \] 
main memory

\[ O(\log\frac{M}{B} \frac{N}{B}) \]
How does priority queue using buffer tree work?

- All buffers on the path from the root to the leftmost leaf must be empty
- For this, Whenever the root is flushed we also flush all buffers down the leftmost path

\[ \Theta(B) \]

\[ \Theta(M/B) \]

main memory

The buffer is not full

\( O(\log_{M/B} \frac{N}{B}) \)
How does priority queue using buffer tree work?

- All buffers on the path from the root to the leftmost leaf must be empty.
- For this, whenever the root is flushed we also flush all buffers down the leftmost path.
How does priority queue using buffer tree work?

- All buffers on the path from the root to the leftmost leaf must be empty
- For this, whenever the root is flushed we also flush all buffers down the leftmost path
I/O Analysis for Priority Queues

- All buffers on the leftmost path are flushed with $O\left(\frac{M}{B} \log \frac{N}{BM} \right)$ I/Os.
- We have $O(M)$ operations with each flush of the root buffer.
- The amortized cost of these extra flushes is $O\left(\frac{1}{B} \log \frac{N}{BM} \right)$ I/Os per operation.
I/O Analysis for Priority Queues

- All buffers on the leftmost path are flushed with $O\left(\frac{M}{B} \log \frac{N}{MB} \right)$ I/Os.
- We have $O(M)$ operations with each flush of the root buffer.
- The amortized cost of these extra flushes is $O\left(\frac{1}{B} \log \frac{N}{MB} \right)$ I/Os per operation.

Results

- Find-minimum queries can be answered on-line without using any I/Os.
- It can shown that is impossible to perform insertion and delete minimums in $o\left(\frac{1}{B} \log \frac{N}{MB} \right)$ I/Os.
- Open problems.
Hashing Based Dictionaries

Hashing Based Dictionaries
Look up with Good Expected Performance

- We will consider *linear probing* and *chaining with separate lists*
- These schemes need only a single hash function $h$ in internal memory
- We assume that any hash function value $h(x)$ is uniformly random
Look up with Good Expected Performance

- We will consider linear probing and chaining with separate lists.
- These schemes need only a single hash function $h$ in internal memory.
- We assume that any hash function value $h(x)$ is uniformly random.

**Load factor $\alpha$**

- $M$ is the number of different addresses are produced by hash function and $N$ is the number of keys.
- $\alpha = \frac{N}{M}$
Linear Probing Example

- $h(k) = k \mod 13$
- Insert keys:
- 18 41 22 44 59 32 31 73
Linear Probing Example

- \( h(k) = k \mod 13 \)
- Insert keys:
- 18 41 22 44 59 32 31 73

![Linear Probing Example Diagram]
Linear Probing Example

- \( h(k) = k \mod 13 \)
- Insert keys:
- 18 41 22 44 59 32 31 73
Linear Probing Example

- $h(k) = k \mod 13$
- Insert keys:
- 18 41 22 44 59 32 31 73
Linear Probing Example

- \( h(k) = k \mod 13 \)
- Insert keys:
- 18 41 22 44 59 32 31 73
Linear Probing Example

- $h(k) = k \mod 13$
- Insert keys:
- 18 41 22 44 59 32 31 73
Linear Probing Example

- $h(k) = k \mod 13$
- Insert keys:
- $18 \ 41 \ 22 \ 44 \ 59 \ 32 \ 31 \ 73$
Linear Probing Example

- $h(k) = k \mod 13$
- Insert keys:
- 18 41 22 44 59 32 31 73

```plaintext
0 1 2 3 4 5 6 7 8 9 10 11 12
0 1 41 4 18 44 59 32 22 31 73
```

Zorieh Soltani (Yazd University)
1. Linear Probing

Operations

The expected average number of I/Os for a lookup is

\[ 1 + (1 - \alpha) - 2\alpha \]

\[ \alpha \leq 1 - \varepsilon \]

and B is not too small =⇒ the expected average is very close to 1.

The probability of using k (more than one) I/Os for a lookup is

\[ 2 - \Omega(B(k - 1)) \]
1. Linear Probing

Operations
- Insertion

The expected average number of I/Os for a lookup is
\[ 1 + (1 - \alpha) - 2 - \Omega(B) \]
where \( \alpha \leq 1 - \varepsilon \) and \( B \) is not too small, which implies that the expected average is very close to 1.

The probability of using \( k \) (more than one) I/Os for a lookup is
\[ 2 - \Omega(B(k-1)) \]
1. Linear Probing

Operations
- Insertion
- Deletion

The expected average number of I/Os for a lookup is $1 + (1 - \alpha) - 2\alpha - \Omega(B)$.

The probability of using $k$ (more than one) I/Os for a lookup is $2 - \Omega(B(k-1))$. 

The number of I/Os for a lookup is very close to 1 if $\alpha \leq 1 - \epsilon$ and $B$ is not too small.
1. Linear Probing

**Operations**

- Insertion
- Deletion
- Lookup

The expected average number of I/Os for a lookup is:

$$1 + (1 - \alpha) - 2^k - \Omega(B)$$

where $$\alpha \leq 1 - \varepsilon$$ and $$B$$ is not too small implies the expected average is very close to 1.

The probability of using $$k$$ (more than one) I/Os for a lookup is:

$$2^{-\Omega(B(k-1))}$$
1. Linear Probing

**Operations**
- Insertion
- Deletion
- Lookup

**The Number of I/Os for a Lookup**
- The expected average number of I/Os for a lookup is 
  \[ 1 + (1 - \alpha)^{-2}2^{-\Omega(B)} \]
- \( \alpha \leq 1 - \varepsilon \) and \( B \) is not too small \( \implies \) the expected average is very close to 1
- The probability of using \( k \) (more than one) I/Os for a lookup is
  \[ 2^{-\Omega(B(k-1))} \]
Chaining works faster than Linear Probing

- Each block in the hash table is the start of a linked list of keys hashing to that block.
- When the pseudo random function works truly, all lists will consist of just a single block.
- The probability that more than $kB$ keys hash to a certain block is at most $e^{-\alpha B(k/\alpha - 1)^2/3}$ (Chernoff bounds).
- The probabilities decrease faster with $k$ than in linear probing.
- If $B$ is large and the load factor is not too high, overflows will be very rare.

\[ P(k > kB) \leq e^{-\alpha B(k/\alpha - 1)^2/3} \]
Making Use of Internal Memory

If sufficient internal memory is available, searching in a dictionary can be done in a single I/O with two approaches:

1. Overflow area
2. Perfect hashing and extendible hashing

Using a Predecessor Dictionary

If we increase internal computation, both internal and external space usage can be made better than of extendible hashing.
1-Making Use of Internal Memory

If sufficient internal memory is available, searching in a dictionary can be done in a single I/O with two approaches:

1. Overflow area
2. Perfect hashing and extendible hashing
1-Making Use of Internal Memory

If sufficient internal memory is available, searching in a dictionary can be done in a single I/O with two approaches:

1. Overflow area
2. Perfect hashing and extendible hashing

2-Using a Predecessor Dictionary

If we increase internal computation, both internal and external space usage can be made better than of extendible hashing
Overflow area

First Idea

- Internal memory for $2^{-\Omega(B)} N$ keys and associated information is available
- Store the keys that cannot be accommodated externally in an internal memory dictionary
- The probability that be more than $2^{-c(\alpha)\Omega(B)} N$ such keys is so small
- If it happens we rehash, choose a new hash function to replace $h$
Second Idea

The overflow area can reside in external memory

For single I/O lookups, internal memory data structures must:

1. Identify blocks that have overflowed
2. Facilitate single I/O lookup of the elements hashing to these blocks
Overflow area (continue)

Second Idea

The overflow area can reside in external memory
For single I/O lookups, internal memory data structures must:

1. Identify blocks that have overflowed
2. Facilitate single I/O lookup of the elements hashing to these blocks

First Task

- It be solved by maintaining a dictionary of overflowing blocks
- This requires $O(2^{-c(\alpha)B} N\log N)$ bits of internal space
Overflow area (continue)

Second Idea
The overflow area can reside in external memory
For single I/O lookups, internal memory data structures must:

1. Identify blocks that have overflown
2. Facilitate single I/O lookup of the elements hashing to these blocks

First Task
It be solved by maintaining a dictionary of overflowing blocks
This requires $O(2^{-c(\alpha)B}N \log N)$ bits of internal space

Second Task
It be solved recursively by a dictionary supporting single I/O lookups
Store a set that with high probability has size $O(2^{-c(\alpha)B}N)$
Mairson introduced a B-perfect hash function

- Hash function $p : K \rightarrow \{1, \ldots, \lceil N/B \rceil \}$
- It maps at most B keys to each block
- A function uses $O(N\log(B)/B)$ bits of internal memory
- If the number of blocks is $\lceil N/B \rceil$, this is the best possible
Mairson introduced a B-perfect hash function

- Hash function $p : K \rightarrow \{1, \ldots, \lceil N/B \rceil\}$
- It maps at most $B$ keys to each block
- A function uses $O(N\log(B)/B)$ bits of internal memory
- If the number of blocks is $\lceil N/B \rceil$, this is the best possible

Disadvantages

1. The time and space needed to evaluate this hash functions is extremely high
2. It seems very difficult to obtain a dynamic version
Extendible Hashing

- Use an internal structure called a *directory*
- Directory is an array of $2^d$ pointers to external blocks
- Random hash function $h : K \rightarrow \{0, 1\}^r$ for $r \geq d$
- Lookup of a key $k$ is performed by using $h(k)_d$
- $h(k)_d$ is $d$ least significant bits of $h(k)$ for determine an entry in the directory
- The parameter $d$ is the smallest number that with it at most $B$ dictionary keys map to the same value under $h(k)_d$
- If $r \geq 3\log N$, such a $d$ exists with high probability, else we rehash it
The Main Results

- Lookups uses a single I/O and constant internal processing time.
- The expected number of directory’s entries is $4 \frac{N}{B} N^{1/B}$.
- If we have $N/B$ blocks $\Rightarrow$ we require $\frac{1}{2} N \log(B)/B + \Theta(N/B)$ bits of internal space (it is close to optimal).
- It can be shown that about 69 percent of the space is utilized.
Extendible Hashing adapts to changes of the key set

- The level of a block is the largest $d' \leq d$ for which all its keys map to the same value under $h_{d'}$.
- Whenever a block at level $d'$ has run full, it is split into two blocks at level $d' + 1$ using $h_{d' + 1}$.
- In case $d' = d$ we first need to double the size of the directory.
- If two blocks at level $d'$ with keys having the same function value under $h_{d'-1}$ contain less than $B$ keys in total, these blocks are merged.
- If no blocks are left at level $d$, the size of the directory is halved.
Lookup Using Two Parallel External Memory Accesses

Two-Way Chaining Scheme

It can be thought of as two chained hashing data structures.

We have two pseudo random hash functions $h_1$ and $h_2$.

Key $x$ reside in either block $h_1(x)$ of hash table one or block $h_2(x)$ of hash table two.

New keys are inserted in the block with the smallest number of keys, with ties broken such that keys go to table one.

Analysis

The probability of an insertion causing an overflow is $\frac{N}{2} \Omega (1 - \alpha)$.

The effect of deletions does not appear to have been analyzed.
Two-Way Chaining Scheme

- It can be thought of as two chained hashing data structures
- We have two pseudo random hash functions $h_1$ and $h_2$
- Key $x$ reside in either block $h_1(x)$ of hash table one or block $h_2(x)$ of hash table two
- New keys are inserted in the block with the smallest number of keys, with ties broken such that keys go to table one
Hashing Based Dictionaries

Lookup Using Two Parallel External Memory Accesses

Two-Way Chaining Scheme

- It can be thought of as two chained hashing data structures
- We have two pseudo random hash functions $h_1$ and $h_2$
- Key $x$ resides in either block $h_1(x)$ of hash table one or block $h_2(x)$ of hash table two
- New keys are inserted in the block with the smallest number of keys, with ties broken such that keys go to table one

Analysis

- The probability of an insertion causing an overflow is $N/2^{\Omega(1-\alpha)B}$
Hashing Based Dictionaries

Lookup Using Two Parallel External Memory Accesses

Two-Way Chaining Scheme

- It can be thought of as two chained hashing data structures
- We have two pseudo random hash functions $h_1$ and $h_2$
- Key $x$ reside in either block $h_1(x)$ of hash table one or block $h_2(x)$ of hash table two
- New keys are inserted in the block with the smallest number of keys, with ties broken such that keys go to table one

Analysis

- The probability of an insertion causing an overflow is $\frac{N}{2^{\Omega(1-\alpha)B}}$
- The effect of deletions does not appear to have been analyzed
Resizing Hash Tables

Keep $\alpha$ in a certain interval to have a good external memory utilization
Resizing Hash Tables

Keep $\alpha$ in a certain interval to have a good external memory utilization

The challenge

Rehash to the new table without an expensive reorganization of the old hash table
Resizing Hash Tables

Keep $\alpha$ in a certain interval to have a good external memory utilization.

**The challenge**
Rehash to the new table without an expensive reorganization of the old hash table.

**The Solution**
- Choosing a new convenient hash function.
- This requires a especial random permutation of the keys.
- For this task we require $\Theta(\frac{N}{B} \log \frac{N}{B})$ I/Os.
- $N = (M/B)^{o(B)} \Rightarrow O(N)$ I/Os.
- $\Theta(N)$ updates between two rehashes.
Resizing Hash Tables Example

$$h(k) = k \mod 4$$

5 Items
Resizing Hash Tables Example

5 Items

\[ h(k) = k \mod 8 \]

Grow the array
Resizing Hash Tables Example
Resizing Hash Tables Example
Resizing Hash Tables Example
Resizing Hash Tables (continue)

Linear Hashing

The Basic Idea for Hashing to a Range of Size $r$

Extract $b = \lceil \log r \rceil$ bits from a mother hash function.

If $b$ bits encode an integer $k$ less than $r$, this is used as the hash value.

Otherwise, the hash function value $k - 2^{b-1}$ is returned.

Expand the size of the hash table by one block (increasing $r$ by one).

All keys that hash to the new block $r+1$ previously hashed to block $r + 1 - 2^{b-1}$.

Decreasing the size of the hash table is done in a symmetric manner.

The Main Problem

When $r$ is not a power of 2, the keys are not mapped uniformly to the range.
Resizing Hash Tables (continue)

Linear Hashing

The Basic Idea for Hashing to a Range of Size $r$

- Extract $b = \lceil \log \rceil$ bits from a mother hash function.
- If $b$ bits encode an integer $k$ less than $r$, this is used as the hash value.
- Otherwise, the hash function value $k - 2^{b-1}$ is returned.
- Expand the size of the hash table by one block (increasing $r$ by one).
- All keys that hash to the new block $r+1$ previously hashed to block $r + 1 - 2^{b-1}$.
- Decreasing the size of the hash table is done in a symmetric manner.
Resizing Hash Tables (continue)

Linear Hashing

The Basic Idea for Hashing to a Range of Size $r$

- Extract $b = \lceil \log \rceil$ bits from a mother hash function
- If $b$ bits encode an integer $k$ less than $r$, this is used as the hash value
- Otherwise the hash function value $k - 2^{b-1}$ is returned
- Expand the size of the hash table by one block (increasing $r$ by one)
- All keys that hash to the new block $r+1$ previously hashed to block $r + 1 - 2^{b-1}$
- Decreasing the size of the hash table is done in a symmetric manner

The Main Problem

When $r$ is not a power of 2, the keys are not mapped uniformly to the range
Dynamization Techniques
The Logarithmic Method

The Problem Must Be Decomposable

Split the set $S$ of elements into disjoint subsets $S_1, \ldots, S_k$

Create a (static) data structure for each of them

Queries on the whole set can be answered by querying each of these data structures

Examples of Decomposable Problems

Dictionaries and Priority Queues
The Problem Must Be **Decomposable**

- Split the set $S$ of elements into disjoint subsets $S_1, \ldots, S_k$
- Create a (static) data structure for each of them
- Queries on the whole set can be answered by querying each of these data structures
The Logarithmic Method

The Problem Must Be **Decomposable**

- Split the set $S$ of elements into disjoint subsets $S_1, ..., S_k$
- Create a (static) data structure for each of them
- Queries on the whole set can be answered by querying each of these data structures

Examples of Decomposable Problems

Dictionaries and Priority Queues
The Logarithmic Method (continue)

Obtain data structures with **insertion** and **query** operations
The Logarithmic Method (continue)

Obtain data structures with **insertion** and **query** operations

The Basic Idea
- Maintain a collection of data structures of different sizes
- Merge periodically a number data structures into one
- keep the number of data structures to be queried low
- In internal memory, the number of data structures is $O(\log N)$
The Logarithmic Method (continue)

Obtain data structures with insertion and query operations

**The Basic Idea**
- Maintain a collection of data structures of different sizes
- Merge periodically a number data structures into one
- Keep the number of data structures to be queried low
- In internal memory, the number of data structures is $O(\log N)$

**The External Memory Version of the Logarithmic Method**
- The number of data structures is decreased to $O(\log_B^N)$
- Insertions are done by rebuilding the first static data structure
- The invariant is that the $i$th data structure should have size no more than $B^i$
- If this size is reached, it is merged with the $(i+1)$st data structure
The Logarithmic Method (continue)

Analysis

- Insert N elements, each element is part of a rebuilding $O(B\log_B^N)$ times
- Building a static data structure for N elements uses $O\left(\frac{N}{B} \log_B^k N\right)$ I/Os
- The total amortized cost of inserting an element is $O(\log_B^{k+1} N)$ I/Os
- Queries take $O(B\log_B^N)$ times more I/Os than queries in the corresponding static data structures
Some data structures for sets support deletions, but do not recover the space occupied by deleted elements.

For example, weak delete.

Keep the number of deleted elements at some fraction of the total number of elements is global rebuilding.
Global Rebuilding

- Some data structures for sets support deletions, but do not recover the space occupied by deleted elements.
- For example, weak delete.
- Keep the number of deleted elements at some fraction of the total number of elements is global rebuilding.

The Main Idea

- In a data structure of $N$ elements, whenever $\alpha N$ elements have been deleted, for some constant $\alpha > 0$, the entire data structure is rebuilt.
- The cost of rebuilding is at most a constant factor higher than the cost of inserting $\alpha N$ elements.
- The amortized cost of global rebuilding can be charged to the insertions of the deleted elements.
خداا
آرامشی عطا فرما تا بی‌پلیرم آنچه که نمی‌توانم تغییر دهم و شهامتی تا تغییر دهم آنچه را که می‌توانم و دانشی که بدون تفاوت آن دو را
دکتر علی شریعتی

Thanks
Have a Good Day