Maintaining linked lists

Standard pointer soup solution: traversing $N$ items costs $\Theta(N)$ I/Os in the worst case.

I/O-efficient solution: maintain that list elements in each disk block are consecutive.

But... insertion $\rightarrow$ everything shifts: $\Theta(N/B)$ I/Os
First solution: allow blocks to be only half full

Insertion: if block not full, simply insert

Insertion: if block full, split into two blocks

In any case, insertion takes $O(1)$ I/Os.

Traversal now takes at most $\lceil 2N/B \rceil = \text{still } \Theta(N/B)$ I/Os.
First solution: allow blocks to be only half full

Deletion: if block remains at least half full, just do it.

Deletion: if block becomes less than half full, steal one item from next/previous block:

...but if stealing would make next/previous block less than full, merge the blocks:

In any case, deletion takes $O(1)$ I/Os
First solution: allow blocks to be only half full

What if there is another data structure with pointers (block numbers) to the list:

Merging two blocks affects $\Theta(B)$ pointers $\rightarrow \Theta(B)$ I/Os in second data structure.
Second solution: allow blocks to be only 1/6 full

Let $B$ be the number of items that fits in a block (after reserving space for pointers).

Insertion: if block full, first move $B/2$ items into a new block; then insert.

Deletion: first delete; then, if block $< 1/6$ full, then:
   if there is a neighbour $\leq 5/6$ full, then merge block into that neighbour,
   otherwise steal $B/3$ items from neighbour.

Claim: keeping $c$ pointers per item up to date, takes $O(c)$ I/Os per insertion/deletion (amortized).

Proof: consider this economic system:
when deleting/inserting an item in a block, we put $c$ euros in that block’s savings account;
when updating a pointer to an item in a block, we take 1 euro from the block’s account.

We will prove that the savings of each block always remain non-negative
($\rightarrow$ at any time, #I/Os spent on pointer updates $\leq c$ times #deletions/insertions so far)
Proof by invariant: savings of block $\geq c \cdot |\#\text{items in block} - B/2|$
Second solution: allow blocks to be only 1/6 full

Let \( \text{load}(K) \) be the number of items stored in block \( K \).
To prove: the invariant \( I(K) \) for each block \( K \): \( \text{savings}(K) \geq c \cdot |\text{load}(K) - B/2| \)

Establishing \( I \) for the first block \( K \) of a list: \( I(K) \) starts to hold after the first \( B/4 \) insertions. (Since no items are moved between blocks until then, the delay is not a problem.)

Maintenance under insertions/deletions in a block \( K \) (without splitting, merging, stealing): the left-hand side of \( I(K) \) goes up by \( c \); the RH-side by at most \( c \); \( \rightarrow I(K) \) is maintained.

Maintenance when splitting a block \( K \), we spend \( cB/2 \) euros on updating pointers to \( B/2 \) items. Thus the LH-side of \( I(K) \) goes down by \( cB/2 \). Since \( \text{load}(K) \) changes from \( B \) to \( B/2 \), the RH-side of \( I(K) \) also goes down by \( cB/2 \), so \( I(K) \) is maintained. We create a new block \( K' \) with \( \text{savings}(K') = 0 \) and \( \text{load}(K') = B/2 \), which estabishes \( I(K') \).

Maintenance when merging a block \( K \) into a block \( K' \): we have \( \text{savings}(K) \geq cB/3 \). We spend at most \( cB/6 \) on updating pointers to \( B/6 \) items, and add the remaining savings to \( \text{savings}(K') \). Thus the LH-side of \( I(K') \) goes up by at least \( cB/6 \), and the RH-side changes by at most \( cB/6 \), so \( I(K') \) is maintained.

Maintenance when a block \( K \) steals from a block \( K' \): we spend \( cB/3 \) from the savings of \( K \) on updating pointers to \( B/3 \) items. In both \( K \) and \( K' \), the absolute value of \( \text{load} - B/2 \) decreases by exactly \( B/3 \). Thus the LH-side of \( I(K) \), and the RH-side of \( I(K) \) and \( I(K') \) all go down by \( cB/3 \), maintaining \( I(K) \) and \( I(K') \).
An I/O-efficient lay-out of a binary search tree on $N$ keys: collect nodes in balanced blocks from the top down.

$B = 1 + \text{number of nodes that fit in a block}$

(+) any root-leaf path (e.g. search in a red-black tree) traversed in $\Theta(\log N B)$ I/Os.

(−) hard to maintain: if insertion/deletion causes rotation at the root, everything shifts
First solution: B-tree, nodes have degree between $B/2$ and $B$.

Define (may be different from literature):

$$B = 1 + \#\text{nodes that fit in a block, and} \#\text{data elements that fit in a leaf}$$

think of blocks as nodes of degree $B$, or leaves with up to $B$ items
(internal structure irrelevant for I/O)
First solution: B-tree, nodes have degree between $B/2$ and $B$.

Define the degree of a leaf as the number of data elements stored in it.

Properties to maintain:

- all nodes have degree at most $B$
- all nodes, except the root, have degree at least $B/2$
- if the data structure stores at least two elements, the root has degree at least two
- all leaves are at the same depth
- all internal nodes store, for each child:
  - the address of the block where the child is stored;
  - the maximum key stored in the subtree rooted at that child

$\rightarrow$ #levels in tree on $N$ elements is at most $\lceil \log_{B/2} N \rceil = \lceil \frac{\log N}{(\log B) - 1} \rceil = O(\log_B N)$;
$\rightarrow$ #levels in tree on $N$ elements is at least $\log_B N = \Omega(\log_B N)$;
$\rightarrow$ the whole tree is stored in $\Theta(N/B)$ blocks;
$\rightarrow$ a (range) query returning $T$ answers takes $\Theta(\log_B N + \text{scan}(T))$ I/Os
First solution: B-tree, nodes have degree between $B/2$ and $B$.

Insertion: walk down to leaf; if full, then split; then insert. Note: a split creates another child for the parent → parent may need to be split → may propagate all the way to root.

Implementation by recursive algorithm $\text{INSERT}(K, x)$ which inserts data element $x$ in block with address $K$, and, in case of a split, returns the address of the new block:

Algorithm $\text{INSERT}(K, x)$:

$\text{newBlock} \leftarrow \text{none}$

if block $K$ is a leaf

then if block $K$ is full, move half of its contents into a newly created block $\text{newBlock}$;

insert $x$ into block $K$ or block $\text{newBlock}$ (depending on key);

else decide in which child $C$ of $K$ to insert $x$

$\text{newChild} \leftarrow \text{INSERT}(C, x)$

if $\text{newChild} \neq \text{none}$

then if block $K$ full, move half of its children into a newly created block $\text{newBlock}$;

insert $\text{newChild}$ into block $K$ or $\text{newBlock}$ (maintaining order);

read maximum keys stored in $C$ and, if applicable $\text{newChild}$,
and store these with the references to $C$ and $\text{newChild}$ in $K$ or $\text{newBlock}$

return $\text{newBlock}$

Usage: call on root; if result not $\text{none}$, then root has been split → create new root.

I/O-efficiency: $\Theta(\log_B N)$ recursive calls that use $O(1)$ I/Os each → $\Theta(\log_B N)$
First solution: B-tree, nodes have degree between $B/2$ and $B$.

Deletion: walk down to leaf and delete; if underfull, then steal or merge.
Note: a merge deletes a child of the parent $\rightarrow$ parent may become less than half full so that it needs to steal or merge $\rightarrow$ may propagate all the way to root.

Implementation: similar to insertion.
If after deletion, root has only one child, then root is deleted and replaced by its child.

I/O-efficiency analysis: similar to insertion.
First solution: B-tree, nodes have degree between $B/2$ and $B$.

What if you want to maintain pointers from children to their parents?

**Algorithm** \( \text{INSERT}(K, x) \):

- newBlock ← none
- if block \( K \) is a leaf
  - then if block \( K \) is full, move half of its contents into a newly created block \( \text{newBlock} \);
    - insert \( x \) into block \( K \) or block \( \text{newBlock} \) (depending on key);
- else decide in which child \( C \) of \( K \) to insert \( x \)
  - newChild ← \( \text{INSERT}(C, x) \)
  - if \( \text{newChild} \neq \text{none} \)
    - then if block \( K \) full, move half of its children into a newly created block \( \text{newBlock} \);
      - insert \( \text{newChild} \) into block \( K \) or \( \text{newBlock} \) (maintaining order);
    - read maximum keys stored in \( C \) and, if applicable \( \text{newChild} \), and store these with the references to \( C \) and \( \text{newChild} \) in \( K \) or \( \text{newBlock} \)
- return \( \text{newBlock} \)

I/O-efficiency: \( \Theta(\log_B N) \) recursive calls that may use \( \Theta(B) \) I/Os each \( \to \Theta(B \log_B N) \)
Second solution: B-tree with degree between $B/6$ and $B$.

What if you want to maintain pointers from children to their parents?

Solution: allow nodes to have degree between $B/6$ and $B$.

Analysis: the height of the tree now becomes at most $\lceil \log_{B/6} N \rceil$, still $\Theta(\log_B N)$, so each update introduces or eliminates at most $\Theta(\log_B N)$ children.

With splitting, stealing and merging, a node and its siblings simply maintain an ordered list of pointers to their children. From ordered list maintenance we know:

if we allow nodes (= blocks) in such a list to be only $1/6$ full,
we can spend $O(1)$ I/Os per item (child) to update pointers when an item is moved,
at an amortized cost of $O(1)$ I/Os per insertion/deletion.

→ we can maintain a search tree with parent pointers such that queries, insertions, and deletions each take $\Theta(\log_B N)$ I/Os (amortized).

Actually, parent pointers are even kept up-to-date at an amortized cost of only $O(1)$ I/Os per insertion/deletion. Idea: let each data element or child inserted/deleted pay at least $B/(B - 6)$ euros to the node in which it is inserted or deleted, instead of one euro. Thus, when children are moved between two siblings $K$ and $K'$ during splitting, merging or stealing, there is an extra amount available of at least $B/6 \cdot (B/(B - 6) - 1) = B/(B - 6)$ euros, beyond what is needed to supplement the savings of $K$ and $K'$ and to update the pointers from the children of $K$ and $K'$. The extra amount can be given to the parent of $K$ and $K'$ to pay for the insertion or deletion of $K$ or $K'$ as a child.
Third solution: weight-balanced B-trees, for if you want more

A weight-balanced B-tree (simplified) has the following properties (assume $B \geq 36$):

- the size of each subtree of $h$ levels is at least $(B/6)^h$ and at most $6(B/6)^h$
  (exception: there is no minimum size for the whole tree)
- all leaves are at the same depth
- all nodes store pointers to their children, and the size and the maximum keys of each child’s subtree

→ each node has degree at most $6(B/6)^h/(B/6)^{h-1} = B$
→ each node (except the root) has degree at least $(B/6)^h/(6(B/6)^{h-1}) = B/36$
→ the height of the tree is still $\Theta(\log_B N)$.

Splitting, merging and stealing is not decided by numbers of nodes, but by their weights:
(weight of a node = number of data elements stored in subtree rooted at that node)

A node is split when its weight would otherwise exceed $6(B/6)^h$;
When a node’s weight drops below $(B/6)^h$, it is
either merged with a predecessor or successor with weight at most $5(B/6)^h$,
or, if this cannot be done, it steals children with total weight between
$2(B/6)^h - 6(B/6)^{h-1} \geq (B/6)^h$ and $2(B/6)^h$ from its predecessor or successor.
Third solution: weight-balanced B-trees, for if you want more

Claim: when a node $\nu$ splits, merges, or steals, we can afford to spend $c$ I/Os per element stored in the subtree rooted at $\nu$ and/or its partner in the split, merge or theft, or $7c/B$ I/Os per element stored in the subtree rooted at the parent of $\nu$, at an amortized cost of $O(c \log_B N)$ I/Os per insertion or deletion.

Hints for a proof: maintain the invariant that $\text{savings}(\nu) \geq 7c|\text{weight}(\nu) - 3(B/6)^h|$. When inserting or deleting, pay $7c$ euros to each node on the search path from the root. Each split, merger or theft on the $h$-th level in the tree (leaf level $= 1$) frees at least $7c(B/6)^h$ euros and affects two siblings of total weight at most $7(B/6)^h$; their parent has weight at most $6(B/6)^{h+1} = B(B/6)^h$.

Implications:
without affecting the amortized $O$-bounds on the I/O-efficiency of queries and updates:

- one can use a linear-time I/O-inefficient algorithm to rebuild and rebalance every subtree involved in a split, merger or theft from scratch;
- one can use a linear-time I/O-efficient algorithm (running in $O(\text{scan}(n))$ I/Os on input of size $n$) to rebuild and rebalance each subtree as soon as a child would need to split, merge or steal.

Caveat: rebuilding must be done in such a way that the total amount of money needed on savings accounts does not increase.
Cache-oblivious search trees

Van-Emde-Boas lay-out:

\[ h = \Theta(\log N) \]

deepth

Cut the three roughly at level \( h/2 \), rounded up to power of two.
Lay-out in memory: first the top above the cut (organized recursively in the same way),
then the subtrees below the cut one by one (each organized recursively in the same way)

Analysis of query time:
At some level of cutting, subtrees have height at most \( \log B \) and at least \( (\log B)/2 \).
Each of these subtrees has size \( \leq B \) and is stored in \( \leq 2 \) blocks (may cross a boundary).
Any root-to-leaf path traverses at most \( \lceil \log N/((\log B)/2) \rceil \) of these subtrees,
in at most \( 2\lceil \log N/((\log B)/2) \rceil = \Theta(\log_B N) \) I/Os.

Maintaining cache-oblivious search trees: much harder than cache-aware search trees