different operations take extra time if you need to pull from further down the cache/disk hierarchy.

Ecobon = h fine

Wᶐ

* sk

tape

disk

L2 cache

L1 cache

registers

CPU

RAM

Eco bon
into pulled in and out as blocks with multiple pieces of data

external memory model

(1988)

- single processor with internal memory storing up to M objects

+ unbounded external memory

all computation in internal
- one I/O operation moves B objects to/from external memory

- want to minimize # I/Os, also called running time

- want to minimize amount of external memory used for data structures

- $N$: # input objects (problem size)

- Assume $2B \leq M < N$

  $m = \lfloor N/B \rfloor$: # blocks in internal mem.

  $n = \lfloor N/B \rfloor$: # blocks for input
Searching $N$ objects stored in $N/B$ blocks of external memory.

Want to find $x$.

External Scan $(x)$:

for $i < 1$ to $n$

read block $i$ into internal

if block $i$ contains $x$

return True

return False

Time is $O(N/B) = O(\log n)$.
If objects are sorted...

I/O for middle block to compare against its element.

Recursively search \( \frac{n}{2} \) blocks.

\[ O(lgn) = O(lg\left(\frac{N}{B}\right)) \] I/Os.
B-tree (a variant of one):

\[ O(n) = O\left(\frac{N}{B}\right) \] block to store objects

searches, insertion, and removal take \( O(\log n) = O(\log \left(\frac{N}{B}\right)) \) time

Objects stored in \( \Theta\left(\frac{N}{B}\right) \) leaves of a rooted tree.

Root has between \( 2 + B \) children.

Internal nodes have \( B/2 \) to \( B \) children.

Nodes contain one fewer search key than \( B \) children.
Search \( (x) \): Load root.

Figure out which pair of keys bound \( x \).

Recursively search subtree based on those keys.

\( x = 30 \)

\[ 1 \ 2 \ 0 \ 4 \ 0 \]

(Except for root), each recursive call decreases search space by a \( \Theta(B) \) factor.

So we load \( O(\log_B (N/B)) \) nodes.

Constant \( 4 \) I/Os per node, so \( \sqrt{\text{true}} \).
Insert + Remove

Just like 2-3-4 tree.
Sorting

Solution 1:
- Build a B-tree one insert at a time.
- Each insert takes $O(\log n)$. 
- $N$ inserts, so $O(N \log n) = O(N \log (N/B))$

Solution 2: (bottom-up)
- total external merge sort
Bottom-up merge sort. (internal)

We'll maintain runs of elements: disjoint contiguous sorted subarray.

Initially, n runs of one element each.

While # runs > 1,

merge disjoint pairs of runs

$O(n) \text{ per iteration, } O(\log n) \text{ iterations, } O(n \log n) \text{ total}$
External mergesort:

1) For each disjoint set of $M$ contiguous objects, load them all, sort, write $O(n)$ I/Os.

Now we have $n/m$ sorted runs of $m$ blocks each.

2) Repeatedly combine groups of $m$ runs by doing an $m$-way merge:

\[\text{output} = \begin{array}{c}
\text{merge} \backslash m \\
\end{array}\]
How to do m-way merge with I/O.

Load first block of each run. We fill up a single "output" block.

Repeatedly remove smallest element of the input block, copy to output block.

When an input block is empty, load the next one of its run.

When output block is full, write it to final output run.
$O(n)$ time for all m-way merges in one iteration.

A blocks reduces by factor m each iteration

So $O(n \log_m (n/m))$

= $O(n \log_m n)$

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

I/Os total

Can't do better with a comparison sort.

Can use sorting to build a B-tree in $O(n \log_m n)$ I/Os total.
1) Make leaves:

2) For each level bottom up:
   Find key to separate B nodes.
   Make a node with those keys & store it.

\[ \# \text{nodes in iter } i = \frac{(N/B)}{B^{\omega/2}} \]

Time proportional to \# nodes
\[ \leq 2 \# \text{leaves} \]
\[ = O(N/B) \text{ (after sorting)} \]