External Memory Data Structures.

Basic Model:
- Data lives on an external-memory.
- Access is via blocks of size $B$.

We measure cost as the number of block transfers.

Dictionary Problem

We want to insert, delete, and search in a dictionary that stores comparable elements.

Lower Bound: Searching requires $\Omega(\log_{B+1} n)$ time.

Proof: Reading a block only has $B+1$ possible outcomes with respect to a query value $x$.

$\Rightarrow$ Search algorithm gives a code whose average length is

$$(\text{average search time}) \cdot \log_2(B+1) \geq \log_2 n$$

$\Rightarrow$ average search time $\geq \frac{\log_2 n}{\log_2(B+1)} = \log_{B+1} n$. 
Static Structure: Use a tree where each node has $B+1$ children, and store a node in $O(1)$ blocks.

Tree has depth $\lceil \log_{B+1} n \rceil$, so following a root-to-leaf search path takes $O(\log_{B+1} n)$ block transfers. QED.

Dynamic structure: B-tree.

- All data is stored in the leaves (and copied in internal nodes)
- All nodes, except root, store between $\lceil B/2 \rceil$ and $2B$ items.
- Root stores at least one item.

Height of tree is at most $\lceil \log_{B+1} n \rceil$, so search time is $O(\log_{B+1} n)$. 
Insertion: Do a search. If leaf is not full then just store new element in leaf.

- If leaf is full then split into 2 blocks of size $B$ and $B+1$.
- Recursively insert head of block into the parent.

Deletion: Do a search and remove element from leaf.

- If leaf contains $< B/2$ elements,
  - borrow from neighbouring leaf,
  - if neighbouring leaf has size $B/2$,
    - merge two leaves to get block of size $B-1$
    - recursively delete from parent.

Analysis: Define potential of a block as

$$\Phi(\text{block}) = C \cdot \frac{|\text{elements in block}| - B}{B}$$

- Amortized cost of splitting and merging blocks is $O$,
- Normal case of insertion/deletion increases potential by $C/B$.

$\Rightarrow$ $n$ insertions/deletions result in $O(\log_B n + n/B)$ block transfers.
Cache-Oblivious Model: Same as regular model, but the value of B is unknown.

- Want a data structure that is efficient for any value of B.

Static Data Structure:

- van Emde Boas layout.

\[
\frac{1}{2} \log n \quad \frac{1}{2} \log n \quad \frac{1}{2} \log n \quad \frac{1}{2} \log n
\]

and so on, recursively.
Levels of detail

- At level 0 we have one big tree of size $n$.
- At level 1 we have trees of size $n^{1/2}$.
- At level 2 we have $n^{1/4}$ trees of size $n^{1/4}$.

Consider coarsest level of detail where subtree size is at most $B$.

- This has subtrees of size $B'$, where

$$B'^{1/2} \leq B' \leq B$$

- These subtrees have height at least $\log B'^{1/2} = \frac{1}{2} \log B$.
- Each subtree is stored contiguously in memory, so occupies at most 2 memory blocks.

- A search proceeds through $\frac{\log n}{\frac{1}{2} \log B}$ subtrees, so it visits $O(\log_B n)$ blocks.

$\Rightarrow$ A search takes $O(\log_B n)$ time, even though we don't know the value of $B$. 
Making it Dynamic

Packed arrays: Store $n$ keys in an array of length $O(n)$
- Support insertion and deletion
- Maintain bounded density:
  - A subarray of length $L$ contains $O(L)$ values.

- Insertion and deletion can be done with $O(\log^2 n / B)$ block updates (amortized).

$$\frac{\log^2 n}{B} < \log B n \quad \text{when} \quad B \geq \log n \log \log n \quad (*)$$

E.g., $B = 1024$, $(*)$ is true for $n \leq 2^{128}$ more than the number of atoms in the universe.

Idea: Treat the array like a binary tree that has tighter and tighter occupancy constraints on its nodes as it approaches the root. Rebuild a subtree (consecutive subarray) when one of its children violates its occupancy constant.
Data Structure: vEB layout on top of a packed array.

Each node in vEB tree stores the maximum of the non-empty values in its subtree.

⇒ Search takes $O(\log_B n)$ time.

Update requires redistributing segments of the packed array.

⇒ Fix values in the vEB tree.

- Requires $O(K/B + \log_B n)$ time.
This structure supports updates in

\[ O\left(\frac{\log^2 n}{B} + \log_B n\right) \] time.

Speeding-Up Updates.

To speed up updates, use induction so that the leaves point to blocks of \( O(\log n) \) elements. Most insertions or deletions only operate on a block, at a cost of \( O(\log n/B) \leq O(\log_B n) \).

One out of every \( O(\log n) \) operations operates on the real tree, at a cost of \( O(\log^2 n/B) \), for an amortized cost of \( O(\log n/B) \) per operation.