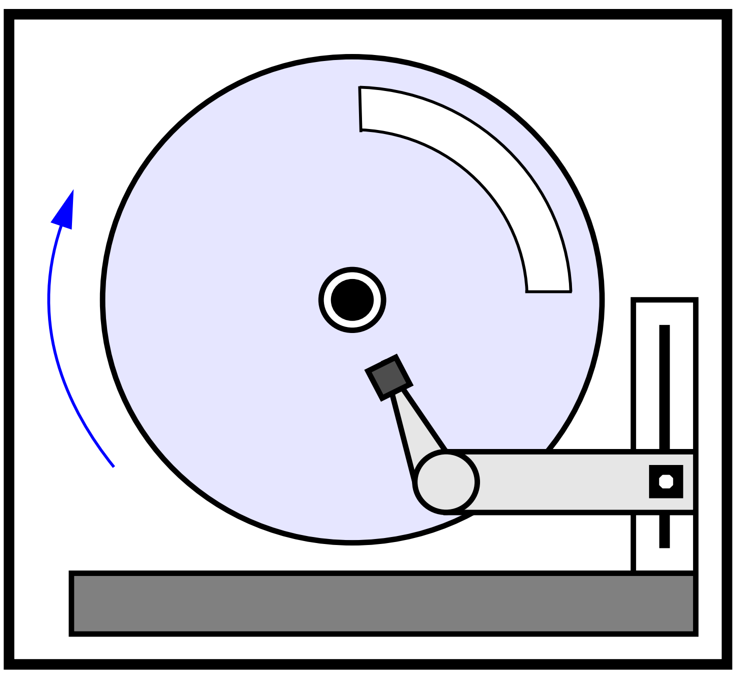
External Memory Computing

- hierarchical memory management
- B-trees
- external sorting

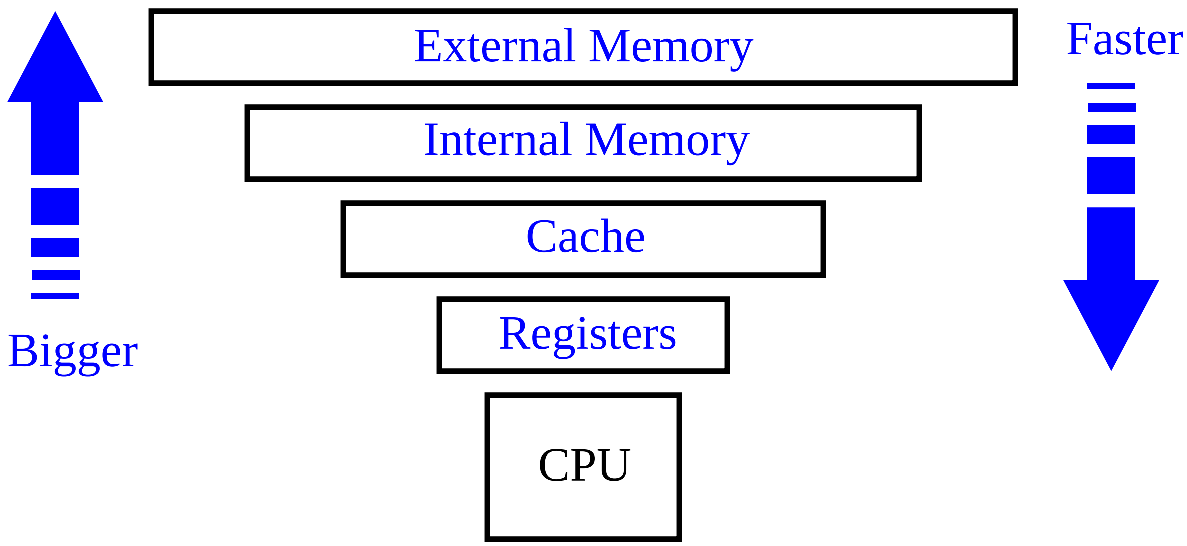


The Memory Hierarchy

- Many problems that modern computers are given to solve (analyzing scientific data, running Win95, etc.) require large amounts of storage.
- In an ideal world, all the necessary information could be stored on chip in the processor's registers, but that would be hideously expensive.
- Instead, computers use a **memory hierarchy** where there is a tradeoff between speed and volume.
- The hierarchy consists of four layers:
  - Registers
  - Cache memory
  - Internal memory (RAM)
  - External memory (Disk)

The Memory Hierarchy (contd.)

- The hierarchy (for a typical workstation):



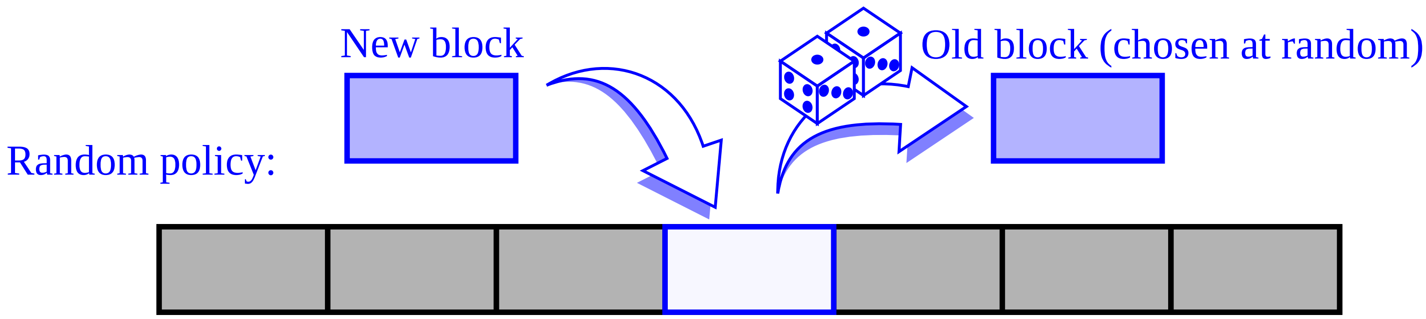
	Access time (CPU cycles)	Volume
Registers:	1 cycle	~2^{10}bytes
Cache:	5 cycles	~2^{20}bytes
Internal:	50 cycles	~2^{26}bytes
External:	2,000,000 cycles	~2^{32}bytes

Caching and Blocking

- Since the performance loss is so great when external memory needs to be accessed, several techniques have been developed to avoid this bottleneck.
- These are based on one of two assumptions about the data:
  - *Temporal Locality*: If data is used once, it will probably be needed again soon after.
  - *Spatial Locality*: If data is used once, the data next to it will probably be needed soon after.
- **Caching** uses virtual memory which is based on Temporal Locality.
  - An address space is provided that is as large as the secondary storage space.
  - When data is requested from secondary storage, it is transfered to primary storage (cached).
- **Blocking** is based on Spatial Locality.
  - When data is requested from secondary storage, a large contiguous block of data is transfered into primary storage.
    (a block of data is paged).

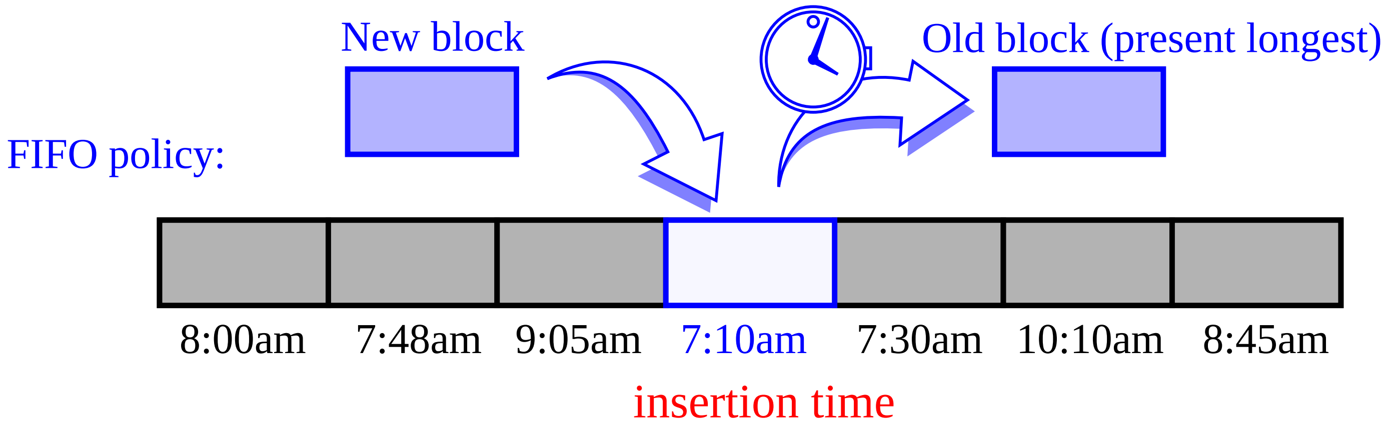
Block Replacement Policies

- We assume we have a **fully associative cache**, that is, a bock from external memory can be placed in any slot of the cache.
- The CPU determines if the virtual memory location accessed is in the cache, and if so where.
- If it is not in the cache the block of external memory, containing the location is transfered into the cache.
- If there are no slots free in the chache, then we must determine which block should be evicted.
- Common policies to determine the block to evict:
  - Random
  - First-In, First-out (FIFO)
  - Least Frequently used (LFU)
  - Least Recently used (LRU)
- **Random** is easy to implement and takes O(1) time



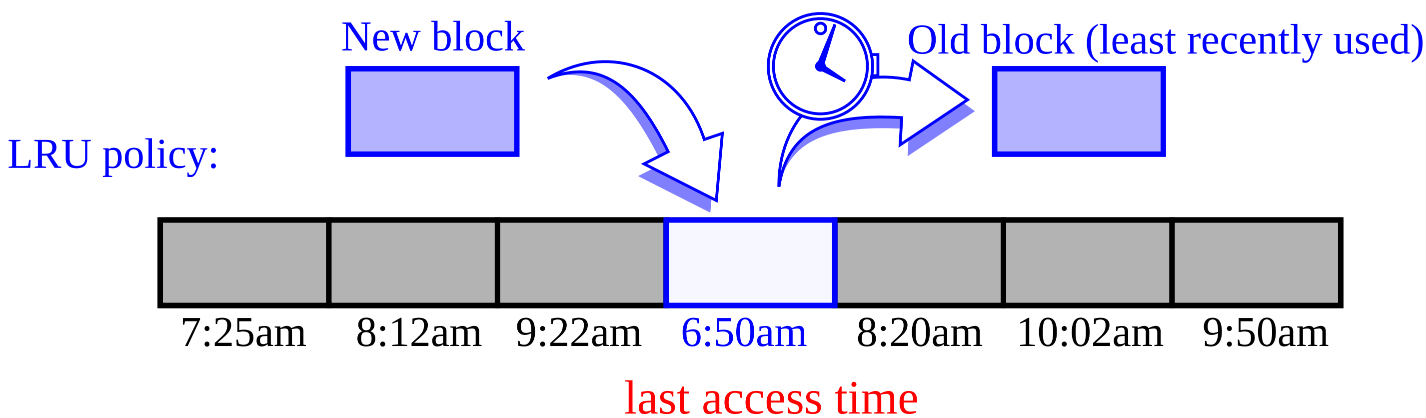
Block Replacement Policies (cont)

- **FIFO** is also easy to to implement, it uses temporal locality and takes O(1) time



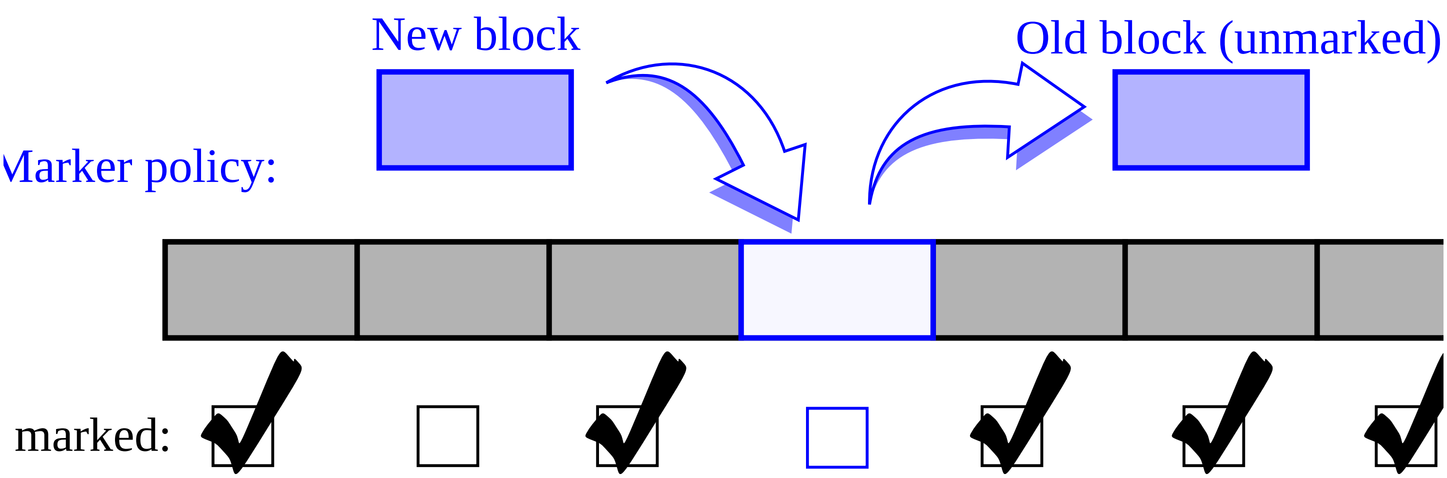
- **LFU** requires more overhead but can still be implemented in O(1) time using a special type of priority queue. But it penalizes recently added blocks.

- **LRU** is the most effective policy in practice. It can be implemented in O(1) time with a special type of priority queue.



The Marker Policy

- mark bit associated with every block in the cache
- if a block in the cache is accessed, it is marked
- if all the blocks become marked, they get all unmarked
- evict a random unmarked block



- this policy is a good approximation of LRU, but is simpler to implement

External Searching

- Let’s look at the problem of implementing a dictionary of a large collection of items that do not fit in primary memory.
- In maintaing a dictionary in external memory we want to minimize the number of times we transfer a block between secondary and primary memory, known as a **disk transfer**, during queries and updates.
- The list-based sequence implentation of a dictionary requires O(n) transfers per query or update.
- The array-based sequence implentation of a dictionary requires O(n/B) transfers per query or update, where B is the size of a block.
- In a binary search tree implentation of a dictionary, in the worst case each node accessed will be in a different block. Thus it requires at least log n transfers per query or update.
- But we can do better ...

(a, b) Trees

- An (a,b) tree is a tree such that:
  - a and b are integers such that $2 \le a \le (b+1)/2$
  - each internal node has at least a children and at most b children
  - all external nodes have the same depth
- Insertion and deletion are similiar to insertion and deletion in (2, 4) trees.
- Properties:
  - the height is $O(\log_a n)$, that is, $O(\log n / \log a)$
  - processing a node takes t(b) time
- A search, insertion, or deletion takes time:

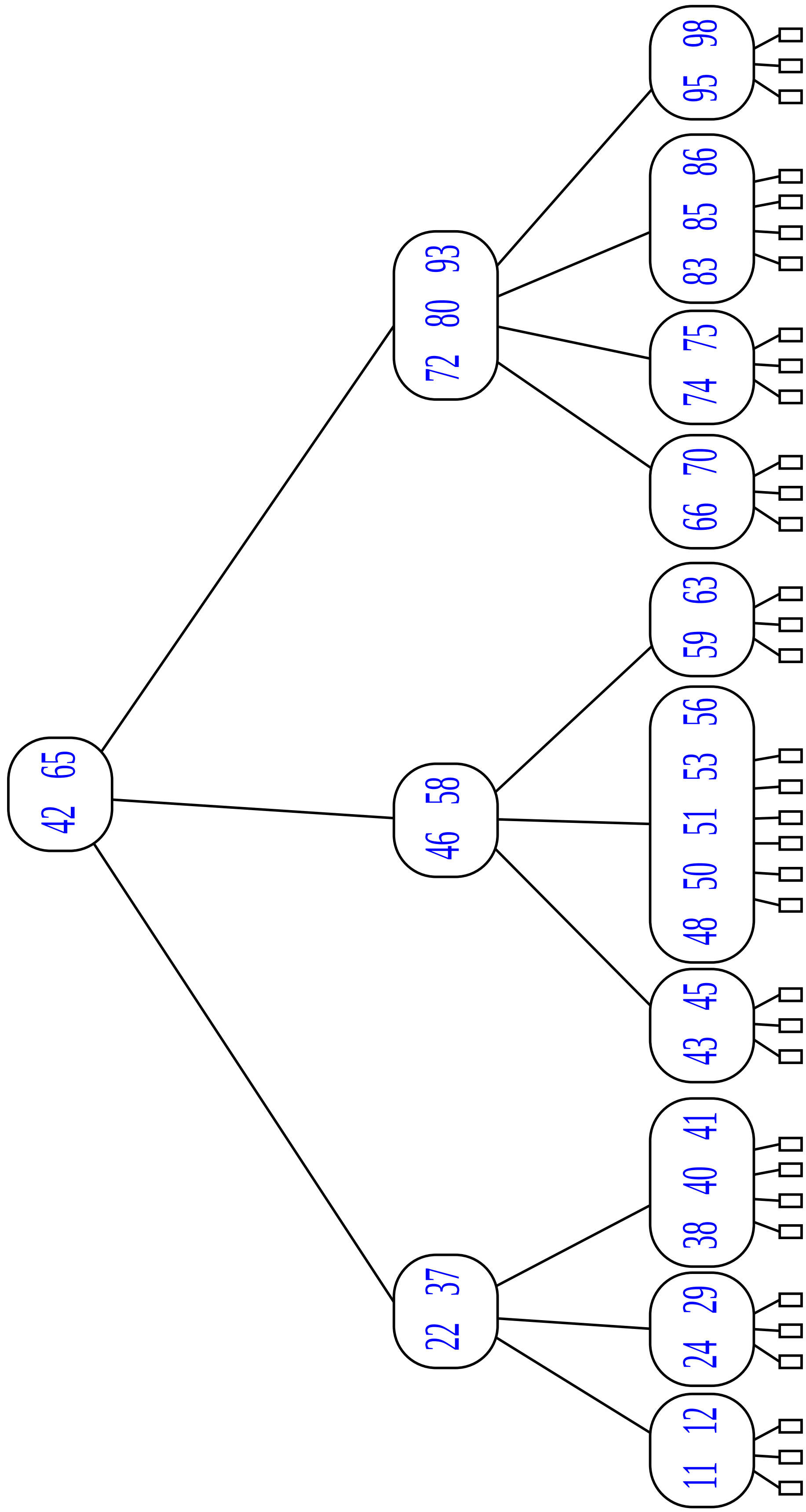
$$O\left(\frac{t(b)}{\log a}\log n\right)$$

and accesses

$$O\left(\frac{\log n}{\log a}\right)$$

nodes (O(1) nodes for each level of the tree).

Example


42 65
22 37
46 58
72 80 93
11 12
24 29
38 40 41
43 45
48 50 51 53 56
59 63
66 70
74 75
83 85 86
95 98

B-Trees

- To minimize disk access we must select values for a and b such that each tree node occupies a single disk block.
- Let B be the size of a block
- A B-tree of order d is an (a,b) tree with a = d/2 and b=d.
- We choose d such that a node fits into a single disk block. This implies a, b, and d are $\Theta(B)$.
- Each search or update requires accessing $O(\log n / \log a)$ nodes.
- Thus, an B-tree requires $O(\log n / \log B)$ disk transfers for any update or search operation.