Algorithms for Memory Hierarchies
Lecture 2

Lecturer: Nodari Sitchianva
Scribes: Robin Rehrmann, Michael Kirsten

Last Time

- External memory (EM) model
- Scan(N): $O(\frac{N}{B})$ I/Os
- Stacks / queues: $O(\frac{1}{B})$ I/Os / elt
- Mergesort: $O(\frac{N}{B} \log_{\frac{M}{B}}(\frac{N}{B}))$ I/Os

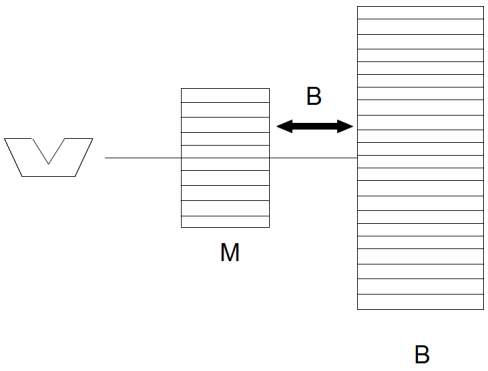

Figure 1: External memory (EM) model. In the EM model we count I/O complexity – number of blocks (of size B) transferred between internal memory of size M and external memory.

Today

- Distribution Sort
- Selection
- Search / B-Trees
- ~~Persistent B-Trees~~

1 Distribution Sort

We start with describing an I/O-efficient algorithm for distribution. Consider Figure 2. It shows some buckets with their boundaries defined as $x_1, x_2, ...$ so that each element y is placed into a bucket whose boundaries are x_i and x_{i+1}, i.e., $x_i \leq y < x_{i+1}$.

We maintain one block for each bucket in memory. When the block becomes full, we write it out to disk, spending an I/O. Since memory is of size M and each block is of size B, we can maintain $\Theta(M/B)$ blocks in memory, and, therefore, perform $\Theta(M/B)$-way distribution.

Thus, we read the input array A and write out the output using one I/O each time we read or write B elements. Since we read and write each element only once, the I/O complexity of distribution is $\mathcal{O}(N/B)$ to read the input and $\mathcal{O}(N/B)$ to write out the output. Thus, the total I/O complexity of performing $\Theta(M/B)$-way distribution is $\mathcal{O}(N/B)$, i.e., linear.

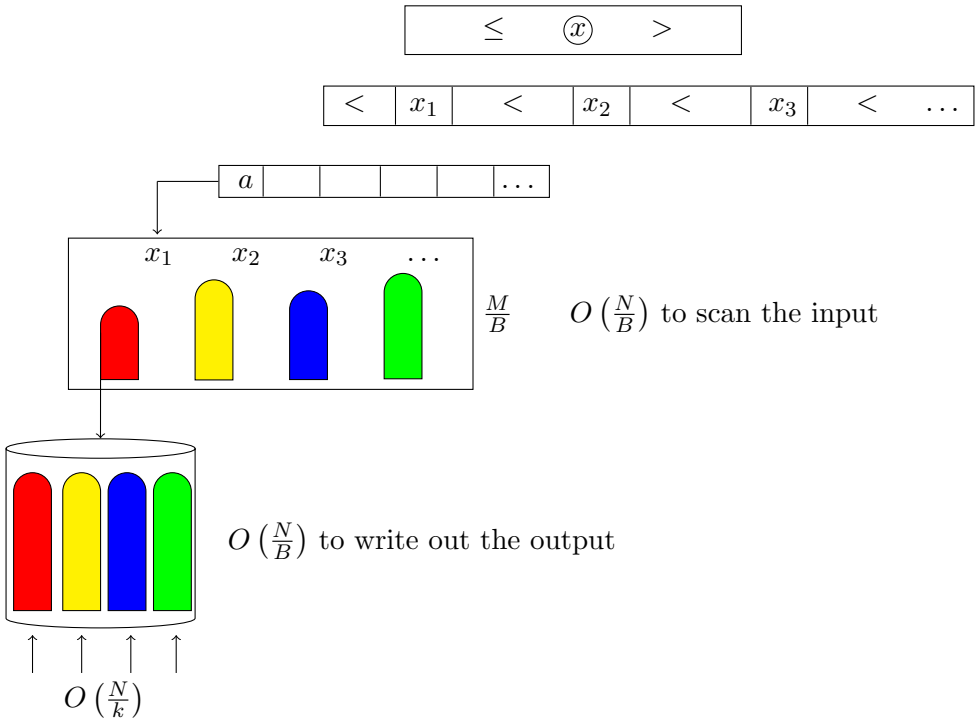

Figure 2: How distribution works

Using $\Theta(M/B)$-way distribution we can sort an array I/O-efficiently as follows. Assume we can pick $t = \Theta(M/B)$ bucket boundaries (we'll call them *pivots*) such that distribution of N elements into these buckets results in each bucket containing $\Theta(N/t)$ elements. Then we can sort using the recursive algorithm in Program 1.

The algorithm recursively distributes the input into t buckets until a bucket fits in

memory, at which point the bucket is loaded into memory and can be sorted using any internal memory sorting algorithm.

```
1  dist_sort(A)
2      if |A| < M
3          Load input into memory and sort it using any
4                      internal-memory sorting algorithm.
5      else
6          find t − 1 pivots
7          distribute the input into buckets B_1, B_2, ..., B_t
8          return {dist_sort(B_1), dist_sort(B_2), ..., dist_sort(B_t)}
```

Program 1: Distribution sort

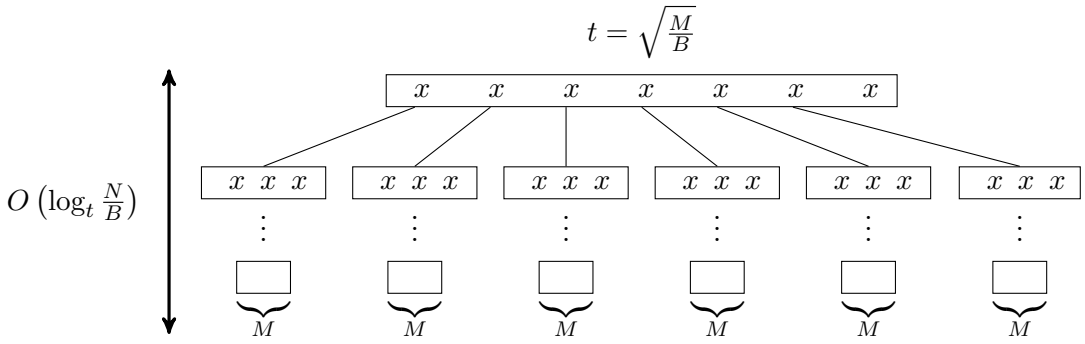

Figure 3: Recursively distribute items into t buckets until each bucket contains at most M elements.

Let's analyze the I/O complexity of the algorithm. In the next section we show how to find t pivots in linear I/O complexity. Thus, for now we can assume that line 6 of the algorithm takes $O(N/B)$ I/Os. As we have shown in the previous section, line 7 also take $O(N/B)$ I/Os as long as $t = \mathcal{O}(M/B)$. Then the I/O complexity of the above algorithm can be described by the following recursion:

$$Q(N) = \begin{cases} tQ(N/t) + O(N/B) & \text{if } N > M \\ O(N/B) & \text{if } N \leq M, \end{cases}$$

which solves to $Q(N) = \mathcal{O}(N/B(1 + \log_t N/M))$. Note, that if $t = \Theta(M/B)$, $Q(N) = \mathcal{O}(N/B \log_{M/B} N/B) = sort(N)$.

The only thing remaining to show is how to pick t pivots so that each bucket is of size $\Theta(N/t)$. In fact, we cannot find $t = \Theta(M/B)$ pivots with this property, however, we will show how to find $t = \Theta(\sqrt{M/B})$ pivots instead. Note, that the I/O complexity of the distribution sorting algorithm is still

$$
\begin{aligned}
Q(N) &= \mathcal{O}\left(\frac{N}{B}\left(1+\log_t \frac{N}{M}\right)\right) \\
&= \mathcal{O}\left(\frac{N}{B}\left(1+\log_{\sqrt{M/B}} \frac{N}{M}\right)\right) \\
&= \mathcal{O}\left(\frac{N}{B}\left(1+2\log_{M/B} \frac{N}{M}\right)\right) \\
&= \mathcal{O}\left(\frac{N}{B}\left(2(1+\log_{M/B} \frac{N}{M})-1\right)\right) \\
&= \mathcal{O}\left(\frac{N}{B}\cdot 2\log_{M/B} \frac{N}{B}\right) \\
&= \mathcal{O}\left(\frac{N}{B}\log_{M/B} \frac{N}{B}\right) \\
&= sort(N)
\end{aligned}
$$

2 Pivot selection

If we choose our pivots randomly, we cannot guarantee that each bucket is of size $O(\frac{N}{t})$ for all inputs. Instead, we show how to find pivots deterministically.

First lets have a look at *single pivot selection*, i.e., how to select a single pivot so that each bucket is of size $\Theta N/2$. Then we show how to select t pivots using *multi pivot selection*.

2.1 Single Pivot Selection

To guarantee that the two buckets are of equal size, we can select the median of the input as the pivot. We will solve a more general problem of finding the i^{th} largest element in an unsorted array.

Consider the simple solution in Program 2. We first sort the array and then return the i^{th} element from the sorted list.

```
1 select(A[1..N], i)
2     sort(A)
3     return(A[i])
```

Program 2: A naive implementation

This naive implementation takes too much time (i.e., $O(N \log_2 N)$) and I/Os (i.e., $O(N/B \log_{M/B} N/B)$) due to costly sorting. So we use a different strategy to solve the selection problem. The pseudocode is presented in Program 3.

```
1  select(A[1..N], i)
2      B = (median(A[1..5]), median(A[6..10]), ..)
3      x = select(B[1..N/5], N/10)
4      A^1[1..l] = A[i] s.t. A[i] ≤ x
5      A^2[1..r] = A[i] s.t. A[i] > x
6
7      if l < i
8          select(A^2[1..r], i − l)
9      else
10         select(A^1[1..l], i)
```

Program 3: Implementation with linear I/Os

First, we split the array A into pieces of size 5 each and consider the medians of each piece (array $\mathcal{B}$ in line 2). We recursively find the median x among these medians (line 3) and partition the original array around this value (lines 4-5). Finally, we recurse within one of the partitions (lines 7-10) adjusting the rank of what we are searching for appropriately.

To analyze the complexity of this algroithm, note that there are two recursive calls in this algorithm:

1) Selecting the median of $\mathcal{B}$.

2) Selecting the i^{th} element in one of the partitions of A.

Before we continue, we prove the following result:

Theorem 1. *The size of the input to the second recursive call is at most* $O(3N/4)$.

Proof. In Program 3 we choose the median of all median elements of all chunks of size *five*. Consider all the chunks of size *five* ordered in increasing order of their median elements (Figure 4) and let x be the median of these medians. Then all medians above x are smaller than x and all medians below x are greater than x. In each chunk, 3 out of 5 of the items are less or equal to its median. Thus, at least half of items in chunks above x are less or equal to x, i.e. at least a quarter of all items are less or equal to x. Analogously at least half the items below x are greater or equal to x, i.e. at least a quarter of all items are greater or equal to x. Thus, after partitioning around x, each partition is of size at least N/4, which implies that each partition is at most of size $3N/4$

□

Thus, the array $\mathcal{B}$ is of size $N/5$ and the size of the partition in the second recursive call is at most $3N/4$. Since the partitioning (distribution around a single pivot) takes linear time and I/O complexity, the time complexity of the above selection algorithm is defined by the recurrence:

$$T(N) = \begin{cases} O(N) + T(\frac{N}{5}) + T(\frac{3}{4}N) = O(N) + T(\frac{19}{20}N) & \text{if } N > c \\ O(1) & \text{if } N \leq c, \text{ for some constant } c \end{cases}$$

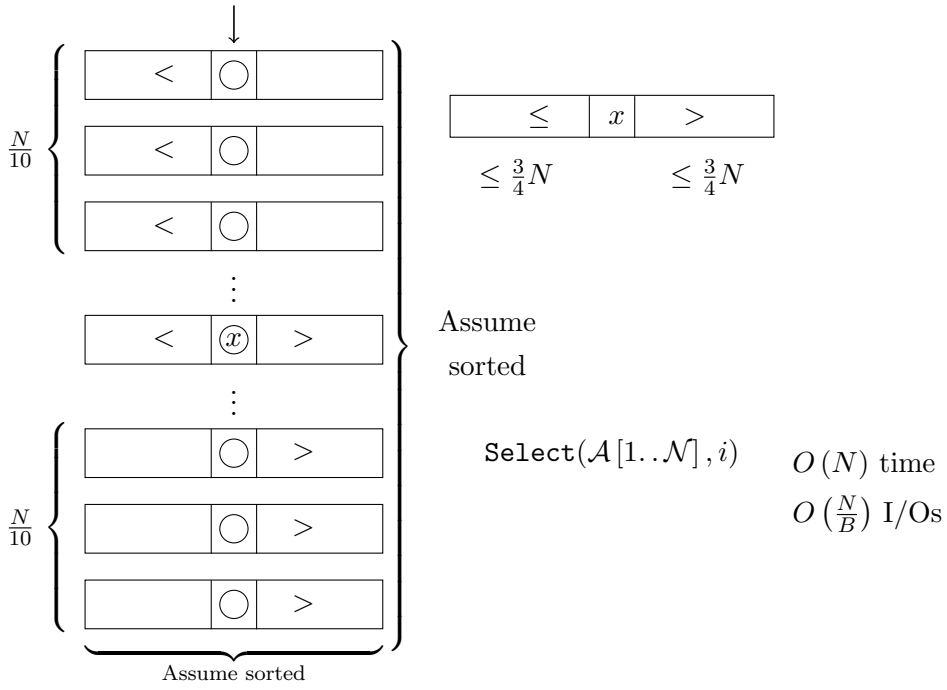

Figure 4: If all elements are sorted, chances to get into the left or right side of x are $\leq \frac{3}{4}N$.

which solves to $T(N) = O(N)$, i.e. linear time complexity.

And the I/O complexity is defined by the recurrence:

$$Q(N) = \begin{cases} O(\frac{N}{B}) + Q(\frac{19}{20}N) & \text{if } N > B \\ O(1) & \text{if } N \leq B \end{cases}$$

which solves to $Q(N) = O(N/B)$, i.e. linear I/O complexity as well.

We will now show that the second recursion step recurses on the array of size at most $O(\frac{3}{4}N)$.

2.2 Multi Pivot select

To select t pivots, if we try to apply the above algorithm t times, the I/O complexity would be $t \cdot O(N/B)$, which is too much. In this section we describe how to select $t = \Theta(\sqrt{M/B})$ pivots in $O(N/B)$ I/Os.

We consider the input array $\mathcal{A}$ of size N as a union of M-sized contiguous subarrays $\mathcal{A}_1, \mathcal{A}_2, \ldots, \mathcal{A}_{N/M}$, where M is the size of memory. Since each such subarray fits in memory, we can load each $\mathcal{A}_i$ into memory and sort it without spending additional I/Os. Next, we pick every $(\frac{t}{4})^{th}$ element from each sorted $\mathcal{A}_i$ and place it into a temporary array $\mathcal{B}$. The total size of $\mathcal{B}$ is, therefore, $\frac{4N}{t}$. Finally, we run the selection algorithm

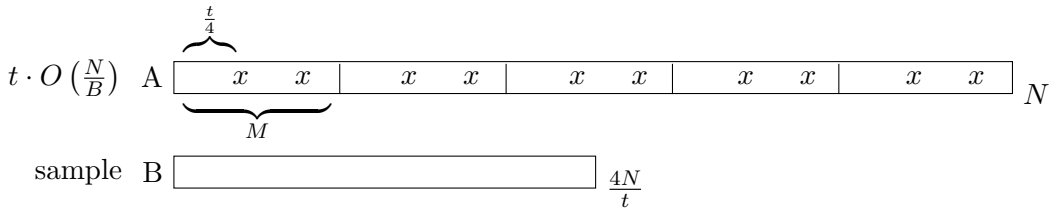

Figure 5: Pick every $(\frac{t}{4})^{th}$ element into $\mathcal{B}$

from the previous section t times to select t pivots $p_1, p_2, \ldots, p_t$ from the elements of $\mathcal{B}$. Each selected pivot p_i is of rank $i \cdot \frac{4N}{t^2}$ among the elements in $\mathcal{B}$, i.e. the pivots would be equally spaced within $\mathcal{B}$ if the array $\mathcal{B}$ were sorted.

Theorem 2. *If $t = \sqrt{M/B}$, the t pivots defined by the above algorithm partition the original input into buckets of size $O(N/t)$.*

Proof. The buckets are defined by every pair of consecutive pivots p_i and p_j. Thus, let us compute the number of elements of the input array $\mathcal{A}$ that fall between two consecutive pivots p_i and p_{i+1}. Let $\mathcal{X}$ be the elements among $\mathcal{B}$ that are at least p_i and at most p_{i+1}, i.e. $\forall x \in \mathcal{X}$, $p_i \leq x \leq p_{i+1}$.

Given how we defined the ranks of p_i and p_{i+1} within $\mathcal{B}$, $|\mathcal{X}| = \frac{4N}{t^2} + 1$.

Then the number of elements of each $\mathcal{A}_j$ that fall between p_i and p_{i+1} is at most $(|\mathcal{A}_j \cap \mathcal{X}| - 1) \cdot t/4 + 2 \cdot t/4$. Then, the total number of elements of $\mathcal{A}$ that fall between p_i and p_{i+1} is at most

$$\begin{aligned} & \sum_{j=1}^{N/M} ((|A_j \cap \mathcal{X}| - 1) \cdot t/4 + 2 \cdot t/4) \\ = \; & \frac{t}{4}\left(\sum_{j=1}^{N/M} |A_j \cap \mathcal{X}|\right) + \frac{N}{M} \cdot \frac{t}{4} \end{aligned}$$

Note that $\sum_{j=1}^{N/M} |A_j \cap \mathcal{X}| = |\mathcal{X}| = \frac{4N}{t^2} + 1$. Therefore, if $t = \sqrt{M/B}$, $M = t^2 B$ and the number of elements that are at least p_i and at most p_{i+1} is at most:

$$\begin{aligned} & \frac{t}{4} \cdot (\frac{4N}{t^2} + 1) + \frac{N}{M} \cdot \frac{t}{4} \\ \leq \; & \frac{N}{t} + \frac{N}{t^2 B} \cdot \frac{t}{4} \\ \leq \; & \frac{2N}{t} = O(N/t) \end{aligned}$$

Since this holds true for each i, and since p_i and p_{i+1} define the boundaries of the i^{th} bucket, the size of each bucket is at most $O(N/t)$. □

Let us analyze the I/O complexity of the multi-pivot selection algorithm. Loading and sorting each of $\mathcal{A}_j$ of size M takes $\sum_{j=1}^{N/M} O(M/B) = O(N/B)$ I/Os. The t runs of the selection algorithm on array $\mathcal{B}$ of size $4N/t$ takes $t \cdot O((4N/t)/B) = O(N/B)$ I/Os. Thus, the whole algorithm takes $O(N/B)$ I/Os.

3 B-tree: an I/O-efficient binary search trees (BST)

There are several balanced binary search trees (BSTs) that support update and search operations in $\mathcal{O}(\log N)$ time:

1. Red Black trees
2. Splay trees
3. AVL-trees
4. (a, b)-trees

The first three BSTs maintain the balance constraint using rotations. Rotations are difficult to make I/O-efficient. Therefore, we will work with the (a, b)-trees today. As we will see later, B-trees and B$^+$-trees are (a, b)-trees with specific values of a and b.

3.1 (a, b)-trees

An (a, b)-tree is a balanced search tree with $2 \leq a \leq \frac{b+1}{2}$. For any node v let $|v|$ denote the number of v's children. Each internal node v except for the root node contains between a and b children, and root contains between 2 and b children, i.e., $a \leq |v| \leq b$ and $2 \leq |root| \leq b$. Each internal node v contains $|v| - 1$ keys $x_1, \ldots, x_{|v|-1}$. Let $w_1, w_2, \ldots, w_{|v|}$ be the children of v. Then all items in the subtree rooted at w_i are greater than x_{i-1} and are less or equal to x_i.

To search for an element y in the (a, b)-tree, we begin at the root node. We find i such that $x_{i-1} < y \leq x_i$ and proceed the search at the child w_i until we are at the leaf node, at which point we either find the item we are looking for or report that it is not stored within the tree.

The pseudocode for inserting a new item into the (a, b)-tree is presented in Program 4.

```
1 insert(x)
2     leaf = search(x)
3     leaf.insert(x)
4     rebalance_insert(leaf)
5
6 rebalance_insert(v)
7     if |v| > b
8         if v is root
```

```
9               x = create_new_root()
10              x.addChild(v)
11          split v into v' and v'' of sizes ⌊|v|/2⌋ and ⌈|v|/2⌉, respectively
12          rebalance_insert(parent(v))
```

Program 4: Implementation of insert and the subroutine rebalance_insert

When inserting an element x into the tree, we first need to find the leaf where the item will be placed and insert it in that leaf. At this point, if the leaf contains less than b elements, we are done. If on the other hand it contains more than b elements, we must split the leaf in two leaf nodes of equal size while inserting an appropriate new routing elements x_i at the parent node. This increases the size of the parent node by one, which in turn might need to be split in two if it becomes larger than b. Thus, we recursively split the nodes that grow larger than b up to the root. If the root grows larger than b, we split it in two and create a new root, which becomes the parent of the two new nodes.

The pseudocode for deleting an item from the (a, b)-tree is presented in Program 5.

```
1  delete(x)
2      leaf = search(x)
3      leaf.remove(x)
4      rebalance_delete(leaf)
5
6  rebalance_delete(v)
7      if v is root and |v| < 2
8          set_root(get_child(v))
9          remove(v)
10     else if |v| == a − 1
11         v' = get_sibling_of(v)
12         v'' = fuse(v, v') // (a − 1 + a ≤ |v''| ≤ a − 1 + b)
13
14         if |v''| > b // (b+1)/2 ≤ |v|, |v'|
15             split v'' into v and v' of sizes ⌊|v''|/2⌋ and ⌈|v''|/2⌉
16     else
17         rebalance_delete(parent(v))
```

Program 5: Implementation of delete and the subroutine rebalance_delete

Again, we first find the leaf containing the element to be deleted. After the element is deleted, if the leaf still contains at least a elements, we are done. If the deletion caused the leaf to contain less than a elements we fuse it with its sibling node into a single node v''. If the new node v'' contains less than b elements, we update the routing elements of the parent node (i.e. remove one of the routing elements x_i). If this update makes the size of the parent drop below a, we recursively rebalancing the parent node. On the other hand, it is possible that the fused node v'' contains more than b children. In this case, we need to split it into two nodes of equal size. Note that the two nodes are at most $\frac{a+b}{2} \leq b$

elements each, because v'' contained at most $a + b$ elements. At the same time, the split happens only if $|v''| > b$, thus, the new nodes contain at least $b/2 > a$ elements each. Thus, we update the routing elements at the parent node and we are done, because all nodes contain between a and b elements.

3.2 B-trees

B-trees are (a, b)-trees with $a = \frac{B}{4}$ and $b = B$. Thus, each node of a B-tree is of size $\Theta(B)$ and, consequently, the height of the tree is $\mathcal{O}(\log_B N)$. It follows that each insertion, deletion and search operation on B-trees takes $\mathcal{O}(\log_B N)$ I/Os to perform.

B^+-trees are B-trees that store the values only at the leaves and the internal nodes store keys just to route the searches within the tree. Note, that B^+-trees take up at most twice the space of B-trees, however, they have the advantage of simpler rebalancing operations.

Application. B-trees are used for indexing data on disks for fast searches and updates, e.g. database queries.