Algorithms for Massive Data Sets

---

**Student name and id:** Alexander R. Johansen (s145706)

**Collaborator name(s) and id(s):** Jonas (s142957)

**Hand-in for week:** 11

---

# 1 Tree Layout in External Memory

The 1.1: $In - OrderLayoutSearch(x)$ Problem can be solved in $O(log(N/B))$ I/O's.
The 1.2: $TreePartitioningLayoutSearch(x)$ Problem can be solved in $O(log(N)/log(B))$ I/O's.
The 1.3: $RecursiveLayoutSearch(x)$ Problem can also be solved in $O(log(N)/log(B))$ I/O's..

## 1.1 In-Order Layout Search(x)

The way in-order traversal folds out the array, causes the need to load a new block at every step through the binary search tree. This happens as all of the elements after an in-order traversal are placed in a monotonically increasing manner, which causes the resulting array to be divided into $O(N/B)$ blocks, e.g.
$[1, 2, 3, 4, 5, 6, 7, 8, 9, 10]$ becomes $B1[1, 2]B2[3, 4]B3[5, 6]B4[7, 8]B5[9, 10]$ an illustration is in appendix A
Binary search requires that one skips half the array on every iteration, thus a new block will be needed to be loaded at every step in the binary search, which would require $log(N)$ I/O's.
However, when the binary search arrives at a node in which the rest of the search is fully contained in the block, we can spare further I/O's and thus arrives at a bound of $O(log(N) - log(B)) = O(log(N/B))$

## 1.2 Tree Partitioning Layout Search(x)

Partitioning the tree into subtrees of size $O(B)$ will allow the requester to require less amounts of blocks loaded, in 1.1 a block would only allow one step down the tree until a block fully containing the rest of the tree was found. The tree partitioning uses the same approach that got the $-log(B)$ in the in-order traversal, and applies it to the entire tree. Thus blocks now represents subtrees, in appendix B I have attached a trivial example.
As the amount of I/O's depends on the amount of B-sized subtrees needed to be traversed, as each subtree allows $O(log(B))$ steps in the tree. Traversing the entire tree would only require $O(log(N)/log(B))$ steps.

### 1.3 Recursive Layout Search(x)

Using the cache-oblivious model, I infer the following recursive layout form the text:

$$IO(N) = \begin{cases} O(1) & \text{if } N \leq B \\ 2IO(2^{log(N)/2})/2 & \text{if } N > B. \end{cases}$$

This recursion explains, that at each step, either N can fit into a block (case 1) or N is too large to fit in a block (case 2) and the algorithm recurses into top and bottom, $T_{top}$ and $T_{bottom}$. It is important to note, that the recursion only creates two trees, as only one of the $k$ different subtrees is required when performing a binary search. Further it is important to notice that the amount of IO's does not happen until the bottom layer is reached, as the cost is not induced before. This makes sense because the recursion is essentially the amount of splits until the path can be covered in some amount of B sub-trees.

**Recursion-tree**

In the Recursion-tree I will use the size of N as a determinant of how many splits to perform, seeing as the size of N reaches B at a given point.

1. $N$
2. $2^{log(N)/2}$
3. $2^{log(2^{log(N)/2})/2} = 2^{log(N)/4}$

...

i. $2^{log(N)/2^i}$

At the level i, as the recurrence grows exponentially, $2IO(..)$ from case 2, I can determine the amount of leaves in my recursion-tree (blocks loaded) to be equal to:

$No.leaves = I/O's = 2^i$

In order to find $i$, I set N in the i'th level to be less than or equal to $B$(as it has reached case 1):

1.$2^{log(N)/2^i} \leq B$

2.$log(N)/2^i \leq log(B)$

3.$log(N)/log(B) \leq 2^i$

4.$log(log(N)/log(B)) \leq i$

Knowing $i$, I can plot the equation $2^i$ being $O(log(N)/log(B))$

# 2 Appendix

## 2.1 A

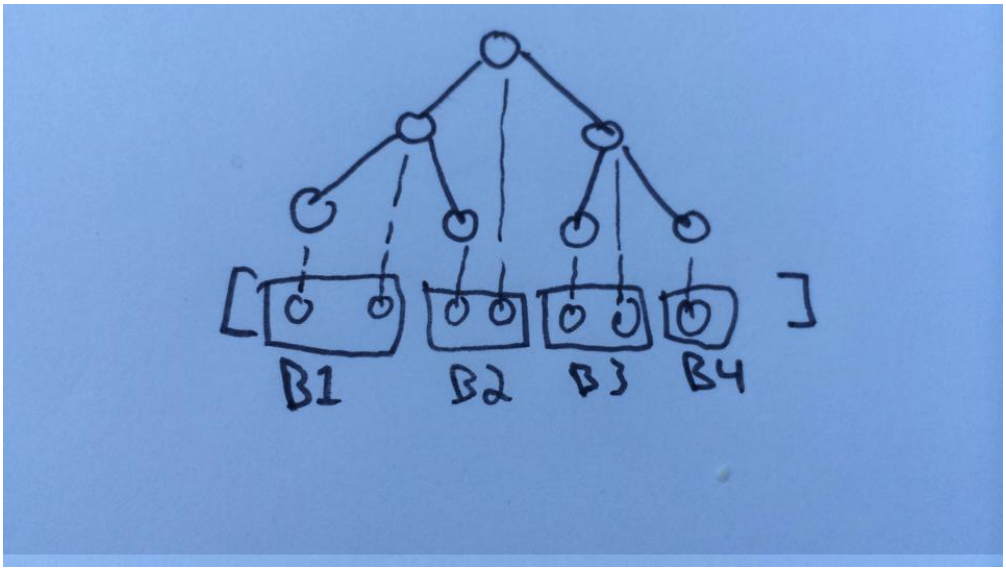


Figure 1: 1.1 In-Order Traversal

## 2.2 B

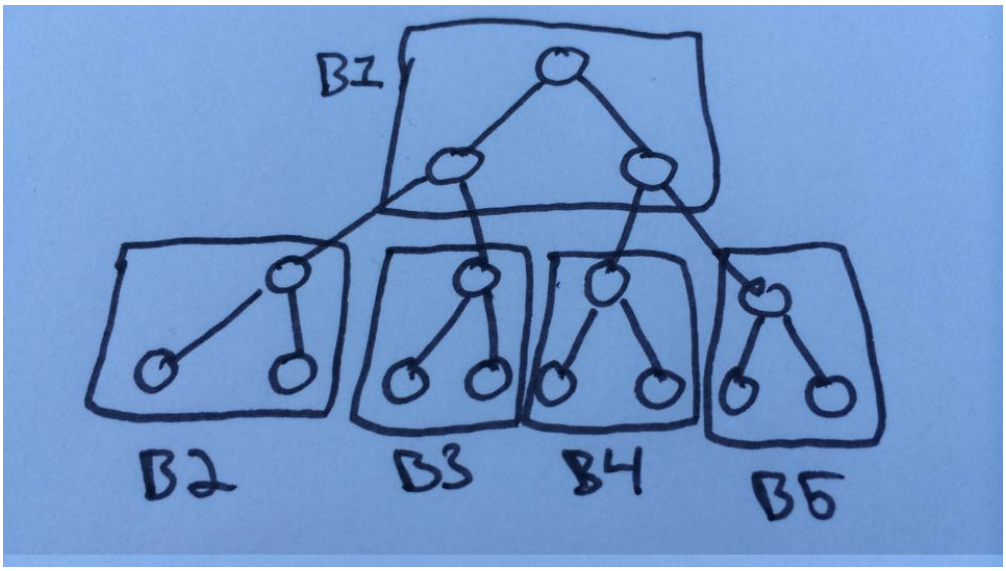


Figure 2: 1.2 Tree Partitioning Layout