External memory implicit searching

Given a static input array A of N keys in the EMM (external memory or cache-aware model), describe how to organize the keys inside A by suitably permuting them during a preprocessing step, so that any subsequent search of a key requires $O(log_B N)$ block transfers using just $O(1)$ memory words of auxiliary storage (besides those necessary to store A). Clearly, the CPU complexity should remain $O(logN)$. Discuss the I/O complexity of the above preprocessing, assuming that it can uses $O(N/B)$ blocks of auxiliary storage. (Note that the additional $O(N/B)$ blocks are employed only during the preprocessing; after that, they are discarded as the search is implicit and thus just O(1) words can be employed.)

SOLUTION

We start with a sorted array A (we can sort it otherwise). First we will build a k-way search tree with exactly the same elements of A (no duplication) i.e. a B-tree. Second we will describe how to store the tree implicitly in an array. Finally we will say how to find the children of a given node without explicit pointers (smell like heap spirit).

Given the block size B, the arity of the tree must be $k = B + 1$. For the construction of the k-way search tree first we need to calculate the height. A complete k-ary tree of height h has 1 node for the first level plus k for the second, plus the square of k for the third etc [1].

$$\begin{aligned} nodes(h) &= \sum_{i=0}^{h} k^i \\ &= k^0 + k^1 + k^2 \cdots + k^h \\ &= \frac{k^{h+1} - 1}{k - 1} \end{aligned}$$

Using

$$(k^0 + k^1 + k^2 \cdots + k^h)(k-1) = (k^1 + k^2 + k^3 \cdots + k^h + k^{h+1}) - (k^0 + k^1 + k^2 \cdots + k^h) = k^h - 1$$

And since each node contains B elements of the array

$$elements(h) = B\frac{k^{h+1} - 1}{k - 1}$$

But in the tree we are going to build we assume that $k - 1 = B$, then

$$elements(h) = B\frac{k^{h+1} - 1}{k - 1} = k^{h+1} - 1$$

The inverse gives us the formula for the height given the number of elements N

$$height(n) = \lceil \log_k(N + 1) - 1 \rceil$$

Before starting the construction of the tree we need one more information about the structure it should have. For reasons that will be clarified later we want a complete tree i.e. one such that every level, except possibly the last, is completely filled, and all nodes in the last level are as far left as possible. Given this shape we want to know how many elements are there in the last level nodes (we name their number L). The calculation is simple since we know the dept of the last level and the number of elements in all the other levels.

$$\begin{aligned} L\ [\textit{last\_level\_elements(N, h)}] &= N - elements(h-1) \\ &= N - (k^h - 1) \end{aligned}$$

[1] Note that we count the height of a tree assuming that a single node tree has height zero, not one.

Now we build the tree, selecting the elements to insert in each node of the tree (B elements for node since every node must be contained in a memory block). We will proceed level by level, from the last to the root (see 1).

For the last level we need exactly L elements, we traverse the sorted array A from left to right selecting B elements and then skipping the next, then taking the B following, then skipping the next and so on, until we select L elements. This way we build the L/B blocks (i.e. nodes) for the last level. It is possible that the last block selected this way remain half empty, we pad with ∞ in that case. Selected elements must be removed from the array.

For the next level the procedure is the same, we scan the remaining array left to right and select blocks of B consecutive elements, than skip the following one and continue, we continue until the end of the array. The procedure is exactly the same for each level until the root, for which we put the remaining B elements.

We number the nodes of the obtained tree level by level, from root to leaves, and left to right. We use this numbering for storing the tree in implicit way (as is usual for heap data structure), with the advantage that pointers are not needed this way. Since we don't use pointers, we must show, given the index of an element, how to find the node which it belongs to and the indexes of the children of the node.

The first problem is trivial: since the number of elements in each node is exactly the block size, the node containing the element at index i is the n-th, with $n = \lfloor i/B \rfloor$ i.e. the block containing i. In the following we call i the index of the element and n the index of the node containing the element.

The problem of finding the children of a given node can be solved in a very elegant way. It comes out that the children of the node n are the nodes $kn + 1$, $kn + 2$, $kn + 3$... $kn + k$. The idea is the following.

Consider that the numbering is such that the index of a node is the number of nodes preceding it in a level by level scan. We call $prec$ the set of preceding nodes of our node n. You can see that multiplying the index by k is equal to taking as number of preceding nodes the number of all the children of the nodes in $prec$, this way we loose the root, but we reach exactly the first child of n. This is because we have the same elements that are in $prec$, all but the root, plus the whole level of n, plus k nodes for each node at the left side of n (see figure 2). We add 1 for the root and we are done.

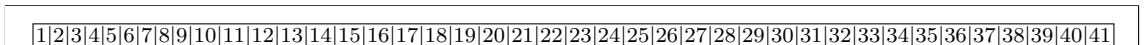
Note that we can do this because the tree is complete. Since we want the indexes of the first element of those nodes we have to multiply those numbers by B, obtaining: $B(kn + 1)$ for the first child, $B(kn + 2)$ for the second, $B(kn + 3)$ for the third etc, until $B(kn + k)$ for the k-th child.

The way we traverse the tree for finding an element e is the usual one.

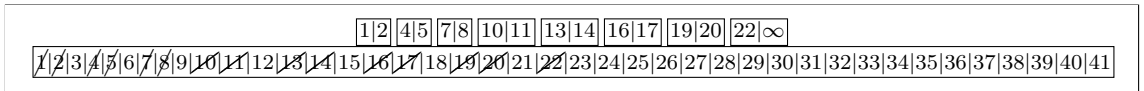
function FIND(e)
  $n \leftarrow 0$ ▷ we start from the root
  while $n < sizeofthearray$ **do**
    load the block in position n
    if e is in the block in position j **then**
      return $n + j$
    else
      find the position j of the first element in the block grater than e, if none $j = B + 1$
      $n \leftarrow B * (k * n + j)$
    end if
  end while
  return -1
end function

The cost of searching is of course $O(\log_B N)$ because at each iteration we divide the tree in which we search by B. The data structure we build occupy exactly the same number of block occupied by A the original array, i.e. $\lceil (N/B) \rceil$ since we store each value only once and all block but one are completely full.

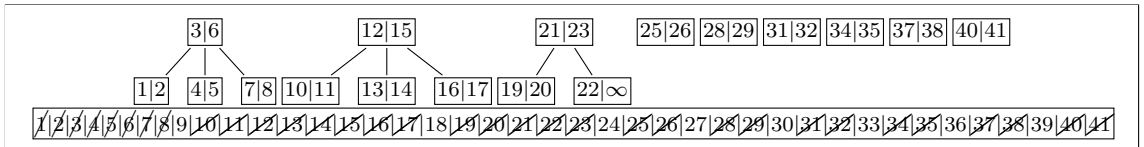
What remain is to discuss the I/O complexity of the construction phase and to show that it doesn't need more than $O(N/B)$ auxiliary storage. For each level of the tree, while building it we must scan the array of remaining elements A block by block, one block is sufficient for this in main memory since when we pass to the next block we don't need the old anymore. We also build the node of each level of the tree in order, from left to right, then we need one block of memory for this, when the node is complete we store it in external memory and start building the next. While building the node of a given level of the tree we also build the new A array for the next iteration, inserting the unused element of the actual A; also this array is build left to right one block at the time, so it require only one block. The procedure require memory size $M \geq 3B + O(1)$.



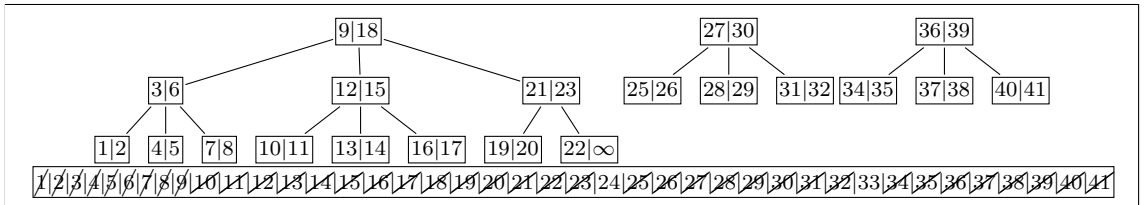
(a) nodes $N = 41$, $B = 2$, $k = 3$, $h = 3$, $L = 15$



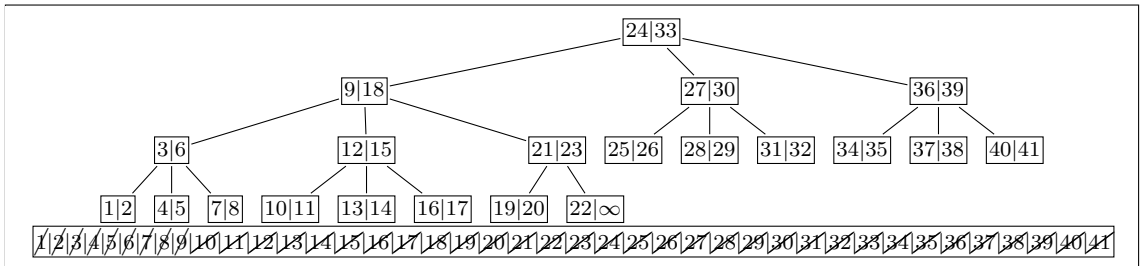
(b) level $i = 3$



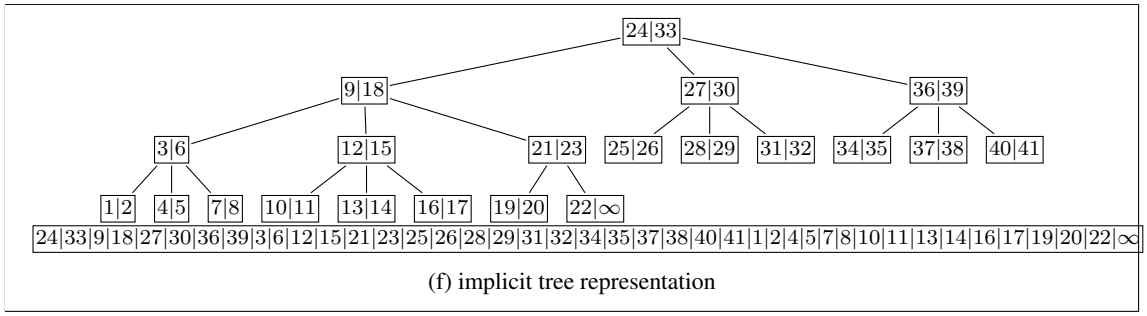
(c) level $i = 2$



(d) level $i = 1$

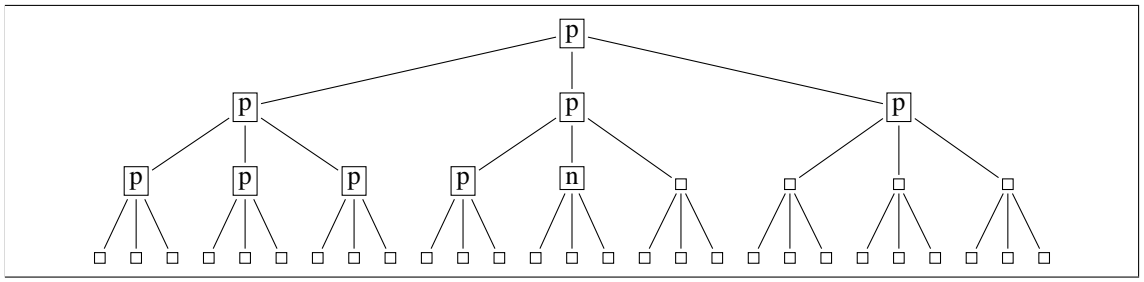


(e) level $i = 0$

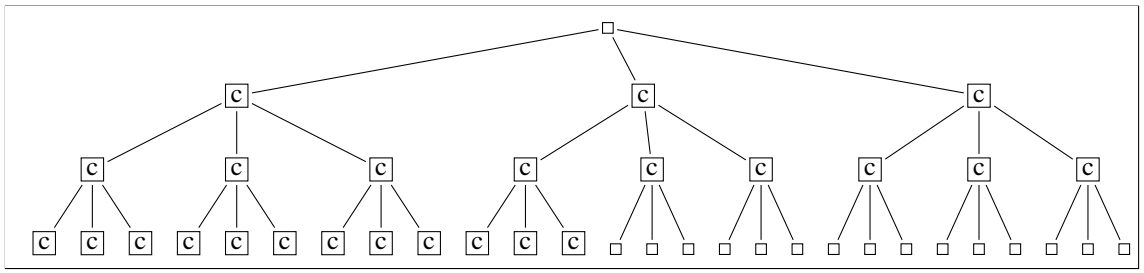


(f) implicit tree representation

Figure 1: tree construction



(a) given node is n, p are elements of prec



(b) given node is n, c are children of elements of prec

Figure 2: number of nodes preceding the children of node n

Note that even if we don't build the nodes of the tree in the exact order we have to store them, for every node, when we build it, we know exactly where to place it since we know the level and the number of previous node in the level. Then we can transform the sorted array A in the implicit tree version without storing in external memory any intermediate form. In external memory we need extra $\lceil N/B \rceil$ blocks for containing the tree we are building (additional with respect to the $\lceil N/B \rceil$ that we need for the original A and that we can continue to use for the array of remaining elements). At the end of the building we remove the space reserved for A and we remain with exactly $\lceil N/B \rceil$.

The I/O cost of the construction phase is then the cost of the creation of the k-ary search tree. For each level we read from memory what remain of the array A and write the exact same number of block in external memory in the form of nodes of the tree or in the form of the new remaining array A.

So the total cost is two times the sum of the length of A divided by the size of the block, for each level. The first iteration is somehow nonstandard, but for all the others we have that the dimension of A decrease and become exactly $|A|/k$ (where we indicate the length of A as $|A|$). We take $\lceil N/B \rceil$ as over bound of the cost of the first iteration. Knowing that the length of A at the beginning of the second iteration is less or equal than N, and that we stop the algorithm when $|A| \leq B$ the total I/O cost is less or equal than

$$\lceil N/B \rceil + 2 \times \sum_{i=0}^{\infty} \frac{N}{Bk^i} = \lceil N/B \rceil + 2N/B \times \sum_{i=0}^{\infty} \frac{1}{k^i} = \lceil N/B \rceil + 2N/B \times \frac{k}{k-1}$$

The last step is given by the rules of geometric series, if x is between 0 and 1 then $\sum_{t=0}^{\infty} x^t = \frac{1}{1-x}$. For concluding, if $k \geq 2$ then the I/O complexity is less or equal than $5N/B$.