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.

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

---

[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 (we can work on indexes in practice, no real work is needed for deleting).

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 $j = B + 1$
            $n \leftarrow B * (k * n + j)$
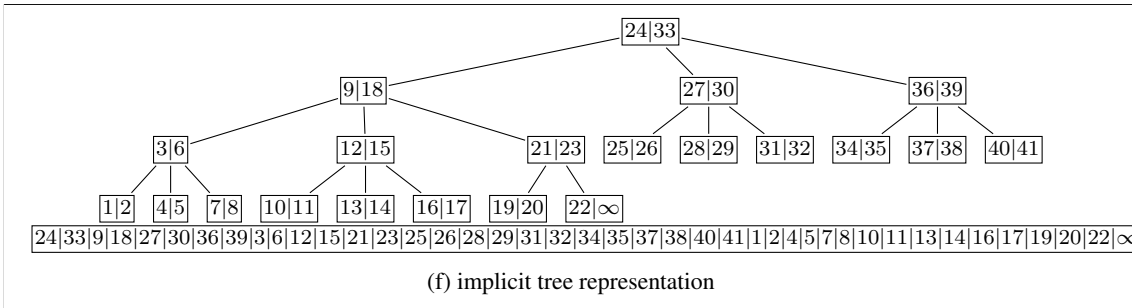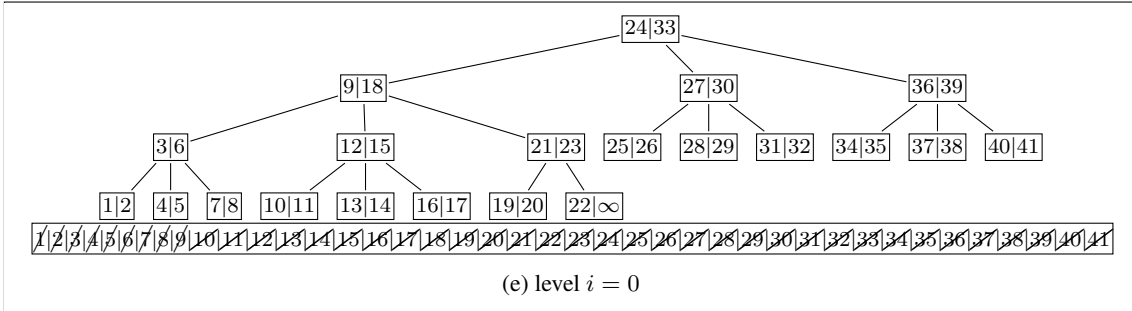   **end if**
    end while
    return -1
end function

(a) 1|2|3|4|5|6|7|8|9|10|11|12|13|14|15|16|17|18|19|20|21|22|23|24|25|26|27|28|29|30|31|32|33|34|35|36|37|38|39|40|41

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

(b) 1|2 4|5 7|8 10|11 13|14 16|17 19|20 22|∞

1|2|3|4|5|6|7|8|9|10|11|12|13|14|15|16|17|18|19|20|21|22|23|24|25|26|27|28|29|30|31|32|33|34|35|36|37|38|39|40|41

(b) level $i = 3$

(c) 3|6   12|15   21|23   25|26 28|29 31|32 34|35 37|38 40|41

1|2 4|5 7|8 10|11 13|14 16|17 19|20 22|∞

1|2|3|4|5|6|7|8|9|10|11|12|13|14|15|16|17|18|19|20|21|22|23|24|25|26|27|28|29|30|31|32|33|34|35|36|37|38|39|40|41

(c) level $i = 2$

(d) 9|18   27|30   36|39

3|6   12|15   21|23   25|26 28|29 31|32 34|35 37|38 40|41

1|2 4|5 7|8 10|11 13|14 16|17 19|20 22|∞

1|2|3|4|5|6|7|8|9|10|11|12|13|14|15|16|17|18|19|20|21|22|23|24|25|26|27|28|29|30|31|32|33|34|35|36|37|38|39|40|41

(d) level $i = 1$

(e) 24|33

9|18   27|30   36|39

3|6   12|15   21|23   25|26 28|29 31|32 34|35 37|38 40|41

1|2 4|5 7|8 10|11 13|14 16|17 19|20 22|∞

1|2|3|4|5|6|7|8|9|10|11|12|13|14|15|16|17|18|19|20|21|22|23|24|25|26|27|28|29|30|31|32|33|34|35|36|37|38|39|40|41

(e) level $i = 0$

(f) 24|33

9|18   27|30   36|39

3|6   12|15   21|23   25|26 28|29 31|32 34|35 37|38 40|41

1|2 4|5 7|8 10|11 13|14 16|17 19|20 22|∞

24|33|9|18|27|30|36|39|3|6|12|15|21|23|25|26|28|29|31|32|34|35|37|38|40|41|1|2|4|5|7|8|10|11|13|14|16|17|19|20|22|∞
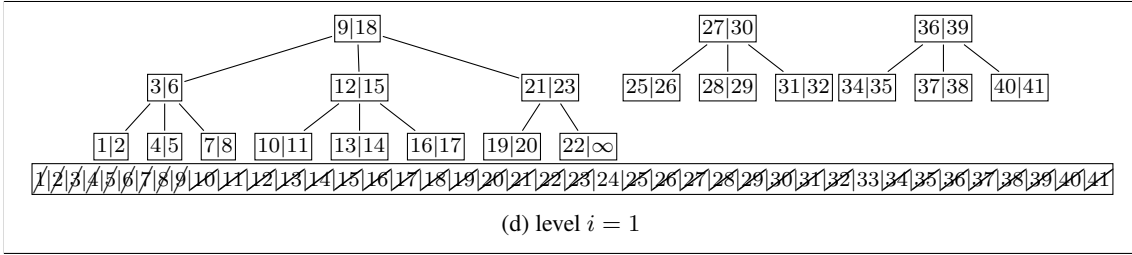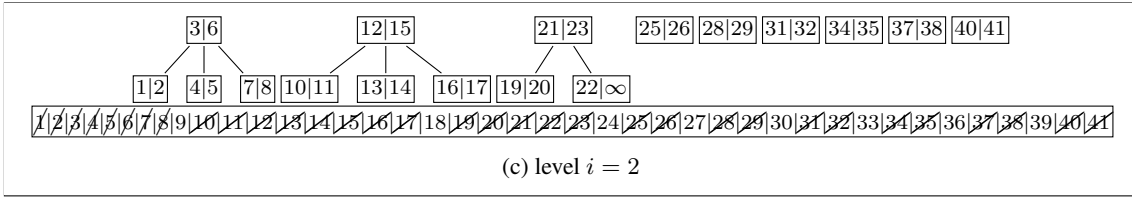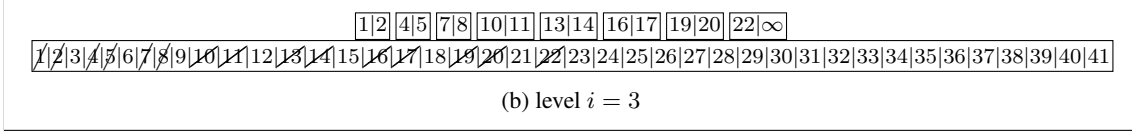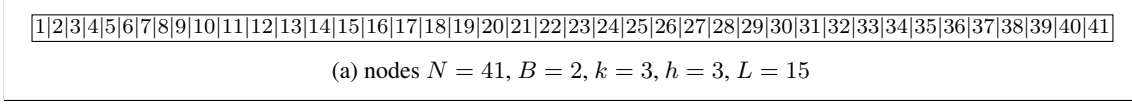
(f) implicit tree representation

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