

1 B-Trees Definition

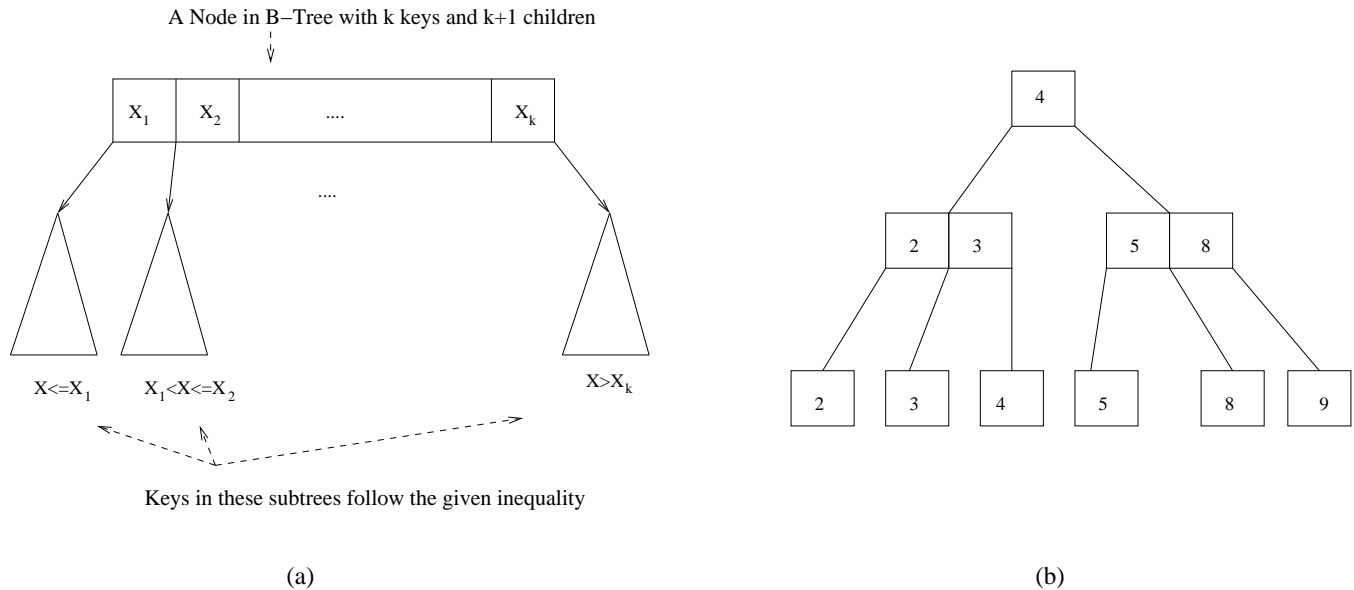


Figure 1: B-Trees.

B-trees are a form of balanced search trees, where each intermediate node has k ($B < k < 2B$) keys, represented by x_1, x_2, \dots, x_k . The keys at a node are ordered, i.e., $x_1 < x_2 < \dots < x_k$. Figure 1, part (a), depicts a node in a B-tree, and Figure 1, part (b), gives an example of a B-tree.

Each node X has $k + 1$ children X_1, X_2, \dots, X_{k+1} with the following properties:

1. Leaves in subtree rooted at X_1 have keys $x \leq x_1$
2. Leaves in subtree rooted at X_i ($1 < i \leq k$) have keys $x_{i-1} < x \leq x_i$
3. Leaves in subtree rooted at X_{k+1} have keys $x > x_k$

2 Priority Queue Implementation Using B-Trees

Priority queues can be implemented using B-trees, where each intermediate node is implemented using an ADT, called a Ranking Dictionary (RD). A RD stores a set $S = \{x_1, x_2, \dots, x_k\}$ of at most $2B - 1$

keys.¹

The following operations can be performed on ADT:

1. *INSERT*(S, y): Insert key y into set S
2. *DELETE*(S, y): Delete key y from set S
3. *SPLIT*(S): Split set S into two equal sized subsets
4. *JOIN*($S1, S2$): Join two sets $S1$ and $S2$ into a single larger set
5. *RANK*(S, y): finds the index of the largest element of S that is less than or equal to y , i.e. $\max\{i | x_i \leq y\}$

We now discuss the implementation of the RD. Our goal is to carry out all the above mentioned operations in constant $O(1)$ time. This is because if we can do so, then the priority queue operations can be carried out in time given by the height of a B-tree. We start with the following lemma.

Lemma 1 *If a RD can be implemented in constant time for all operations, then there is a priority queue in which operations take $O(\frac{\log n}{\log t})$ time.*

Proof: The priority queue is implemented using a B-tree in which each intermediate node is a RD. Therefore, operations on priority queue simply take time given by the height of the tree, which is $O(\log_t n)$, i.e., $O(\frac{\log n}{\log t})$. ■

Now we proceed to show that a RD can be implemented in constant time, and the following lemma gives us a result in this direction.

Lemma 2 *If the word size is at least $\Omega(t * \log m)$, then a RD for sets of size $O(t)$ over a universe of size m can be implemented using $O(1)$ words of memory, and all operations run in $O(1)$ time.*

Proof: We first show that RD takes up $O(1)$ words of memory of size $O(t * \log m)$.

$S = \{x_1, x_2, \dots, x_k\}$, where $k \leq 2t$. Let each x_i be represented using $l (= \lceil \log m \rceil)$ bits.

Also, we represent S , written as $Rep(S)$, as follows,

$$Rep(S) = 0x_10x_2 \dots 0x_k(01^l)^{2t-k}$$

“0”s above are used for padding and act as delimiters. They also act as placeholders for later insertions. In $Rep(S)$, we add string 01^l repetitively, which are the largest key values that can be added to S .

Now storing both S and k require $2t * (\log m) + (\log 2t)$ bits, i.e. they require $O(1)$ words.

After showing that both S and k can be stored in $O(1)$ words, we assume that all word operations take constant time (some techniques for doing so were discussed in the previous lecture). We now look at individual operations on S and show that they run in constant time.

SPLIT(S): Input to the split operation is S

The following steps are carried out -

¹In our discussion here, we assume the universe U to be fixed, although we will see that the size of universe does not play a very important role now.

1. First copy S to S'
2. Create two masks M_1 and M_2 , where $M_1 = 1^{(l+1)t}0^{(l+1)t}$ and $M_2 = 0^{(l+1)t}1^{(l+1)t}$
3. Bitwise AND of M_1 and S removes the greater half of keys in S
4. Bitwise AND of M_2 and S' removes the smaller half of keys in S'
5. Left-shift of S' by $(l+1)t$ bits move keys to the left. Also, do bitwise XOR with $00\dots001^l\dots01^l$ to get the correct representation of the resulting set

Output of the split operation is S and S' . Since, the bitwise AND, XOR, and SHIFT operations on words can be done in constant time, the SPLIT operation runs in constant time.

The JOIN operation is similar (and simpler) to that of SPLIT operation and again runs in constant time. Also, INSERT and DELETE relies on SPLIT and JOIN operations, and so they too can run in constant time. The only remaining operation is RANK.

$RANK(S, y)$: Input to the rank operation is S and y . The following steps are carried out -

1. Multiply y by $(0^l)^{2t}$. The output is the following string: $Y = 0y0y\dots0y$, with $0y$ repeated $2t$ times.
2. Obtain $1y1y\dots1y$ (we do this to carry out the subtraction in the next step without having to worry about the carries, since otherwise if we have 0s then subtraction can cause ripple carry affect). This is done by ORing Y with $(10^l)^{2t}$. Call the resulting output Z , i.e. $Z = Y \vee (10^l)^{2t}$
3. Let $X = 0x_10x_2\dots0x_k01^l\dots$
4. Obtain R , where $R = Z - X = a_1r_1a_2r_2\dots a_kr_k\dots a_{2t}r_{2t}$
5. If $y \geq x_i$, then $1y - 0x_i = 1r_i$, and $a_i = 1$
if $y < x_i$, then $1y - 0x_i = 0r_i$, and $a_i = 0$
6. If $y \neq 1^l$, then counting number of $a_i = 1$ gives number of x_i such that $x_i \leq y$, i.e. the rank of y
if $y = 1^l$, then all a_i 's would be zero, and so return k (which tells that y is larger than all the keys in S)

The next problem is to count the number of a_i s equal to 1. This requires the following:

1. Zero out the r_i 's
2. Sequence of shifts and multiply (see Figure 2 below)

This takes constant number of steps given shifts and multiply are constant time. Thus, one can see that RANK operation can also be carried out in constant time. This completes the proof. ■

Other operations, such as *INSERT*, *DELETE*, and *MEMBER*, call $i = RANK(S, y)$ and compare y with x_i . Therefore, all these operations take constant time also.

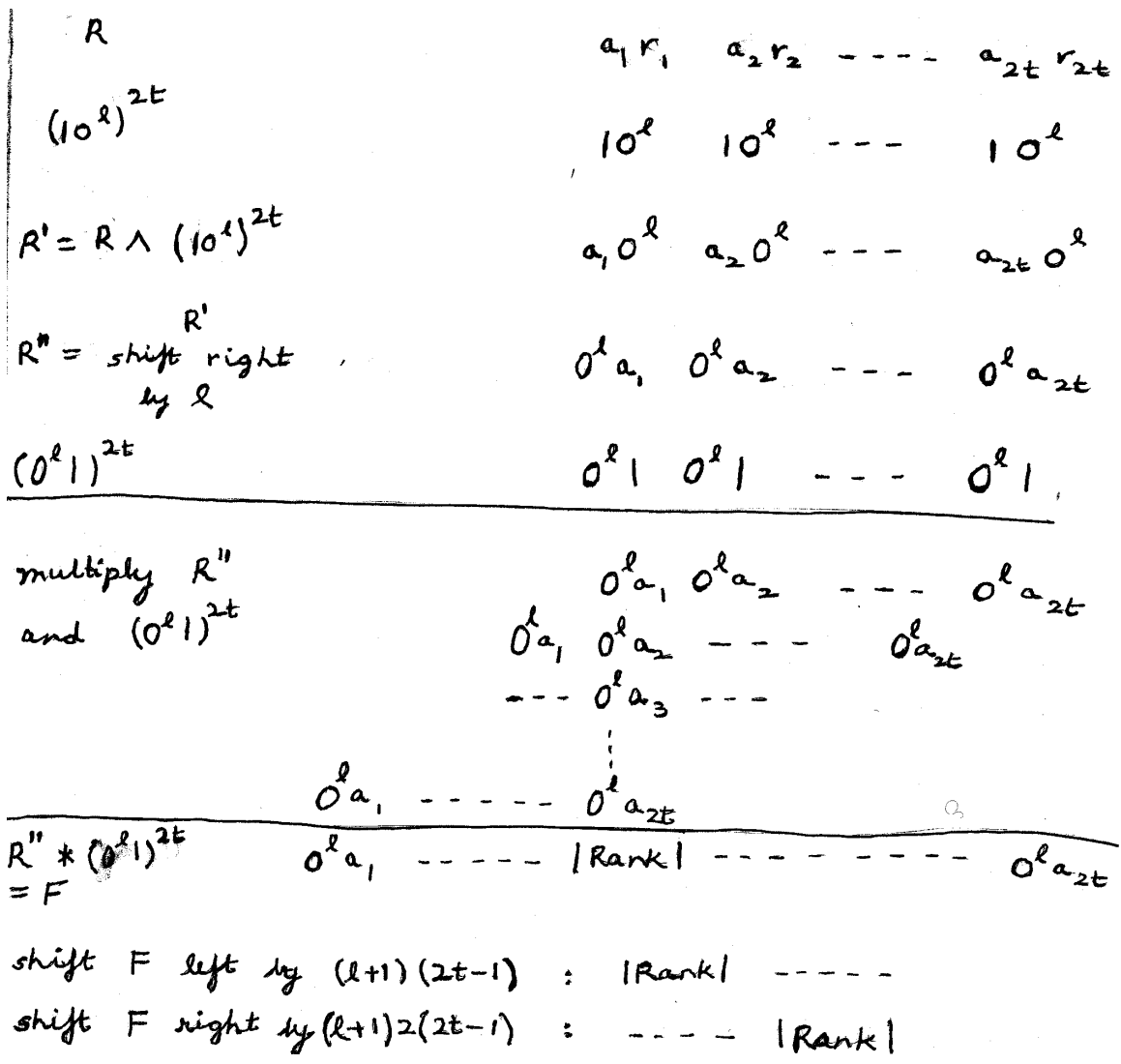


Figure 2: Series of shift and multiply operations.

3 $\theta(\log n / \log \log n)$ Implementation of Priority Queues

Note: We assume that U is large compared to n , however, still any element of U can be represented using constant number of words. Thus, the storage space required for any element is $O(1)$.

Using B-trees, priority queue operations can be carried out in $O(\frac{\log n}{\log t})$ time. It is also possible to have implementations of priority queues that run in $\theta(\log n / \log \log n)$ time. One way to do so is to use fusion trees, which are defined next.

Fusion Trees (Fredman and Willard [1]): Fusion trees are B-trees with branching factor (B) equal to $(\log n)^{1/5}$. This gives us $O(\frac{\log n}{\log \log n})$ run time for priority queue operations.

Claim 3 *Height (h) of a fusion tree is $\theta(\log n / \log \log n)$.*

Proof: Total number of nodes in a tree, denoted by n , are $n = B^h$

$$n = ((\log n)^{1/5})^h \Rightarrow \log n = \frac{h}{5} * (\log \log n)$$

$$\text{or } h = \theta(\log n / \log \log n). \quad \blacksquare$$

In order to get constant time work at each node, we need to find where query y fits among $(\log n)^{1/5}$ keys (each key is assumed to take either a single word or constant number of words). For this convert all $\theta(B)$ keys at each node into binary strings, and view them as paths (as done in binary tries earlier). For, example, consider some four keys in S , $x_0 = 0, x_1 = 17, x_2 = 21, x_3 = 23$. Now representing these four key values require five bits, however, these four keys can be distinguished from each other using (at least) two bits.

In general, if we construct a binary trie using the key values (total k of them) in S , we denote the bit positions corresponding to the branching levels by b_0, b_1, \dots, b_{r-1} , where $b_0 < b_1 < \dots < b_{r-1}$, and $r \leq k - 1$. Using “Perfect Sketch” (defined below), k ($\log m$)-bit strings can be represented using k r -bit strings. Since, $r \leq k$, number of bits required to represent S becomes $k^2 = \theta(B^2) = \theta((\log n)^{2/5})$ (refer to Figure 3 where the number of different branching levels, equal to 3, can be used to differentiate the four key values). And this value is less than a word size. Thus, node operations can be carried out in constant time.

Perfect Sketch (x): The perfect sketch of x is the r -bit vector where the i^{th} -entry is b_i^{th} bit of x . (It is possible to obtain a perfect sketch when the branching factor is $(\log n)^{1/5}$).

Perfect sketch can be difficult to obtain, so instead can use “almost perfect sketch”. In almost perfect sketch, we use bit-padding to make node size equal to $\theta(B^4) = O(\log n)^{\frac{4}{5}}$ bits, which is less than a word size. Thus, constant time operation on each node can still be performed.

The query time using almost perfect sketch on fusion trees is then simply the height of the tree = $\theta(h) = O(\frac{\log n}{\log \log n})$. However, updates can still be expensive since they involve potential re-writing of the sketches, and the time taken is linear in the length of the sketch. Since this time is $\theta(B^4)$, this is not good.

In order to enable fast updates, we add another feature to fusion trees - at each leaf of a fusion tree add a pointer to a regular B-tree (referred to as a subtree) with constant branching. (For amortized

analysis to work, we should have a size of B-tree to be $\theta(B^4)$. The modified fusion tree is depicted in Figure 4.

Now changes to main fusion tree costs $O(B^4)$ amortized time, and time to update the subtree is constant $O(1)$. So, since $\theta(B^4)$ updates in a particular subtree has amortized cost of $\theta(B^4)$, the amortized update complexity is thus constant. Moreover, the query time is same as in the initial fusion tree. This is because additional query work that is to be done inside subtrees is $\log(B^4) = \theta(\log \log n)$, which is asymptotically less than $O(\frac{\log n}{\log \log n})$.

As part of our next step, the goal is to make fusion trees run in $O(\sqrt{\log n})$. For this we use *Van Emde Boas* tree structure, which gives a query time of $O(\log \log m)$. We can combine this tree structure with fusion trees to get the run time of $O(\sqrt{\log n})$. Two cases are possible -

Case 1: if $(\log \log m) \leq (\sqrt{\log n}) \Leftrightarrow (\log \log m)^2 \leq (\log n) \leq n \Leftrightarrow n \geq 2^{(\log \log m)^2}$.

Thus, if $n \geq 2^{(\log \log m)^2}$, then this tree structure gives us the required query time for fusion trees.

Case 2: if $n \geq 2^{(\log \log m)^2}$

$\Rightarrow (\sqrt{\log n}) \leq (\log \log m)$

$2^{\frac{1}{5} * (\sqrt{\log n})} \leq 2^{\frac{1}{5} * (\log \log m)} = (\log m)^{\frac{1}{5}}$

Using the above derived relationship, we utilize the fact that it is okay for fusion tree to have branching factor $(\log m)^{\frac{1}{5}}$. We have,

$$n = B^h = (\log m)^{\frac{h}{5}} \geq 2^{\frac{h}{5} * (\sqrt{\log n})}$$

$$\Rightarrow (\log n) \geq \frac{h}{5} * (\sqrt{\log n})$$

$$\Rightarrow h \leq \frac{5(\log n)}{(\sqrt{\log n})} = \theta(\sqrt{\log n})$$

Since the search time is dependent on h , we thus obtain a search time of $\theta(\sqrt{\log n})$.

4 Q-Heaps

Q-heap data structure (Fredman and Willard JCSS [2]) is an implementation of the priority queue abstract data type. This data structure can perform the priority queue operations in time $O(1)$, provided that the number of elements in the priority queue is small enough (although the universe might be large). Specifically, the constant run time is achieved if $B < \frac{w}{\lceil \log_2 w \rceil} + 1$, where B is the number of elements or keys stored in a Q-heap and w is the word size. Thus, we use Q-heaps to implement a node in a B-tree or fusion tree.

To illustrate the Q-heap data structure, we define set S , such that

$$S = \{x_1, x_2, \dots, x_k\}, \text{ where } x_1 x_2 < \dots < x_k \text{ and } k \leq B$$

Also, let c_i be the MSB position on which x_i and x_{i+1} differ. For example, if $x_1 = 33$, $x_2 = 47$, $x_3 = 53$, and $x_4 = 54$, then $c_1 = 3$, $c_2 = 4$, and $c_3 = 1$ (as shown in Figure 5).

We also define c_j , where $c_j = \max c_i, 1 \leq i \leq k - 1$

Claim 4 j is unique, i.e., c_1, c_2, \dots, c_{k-1} has a unique maxima.

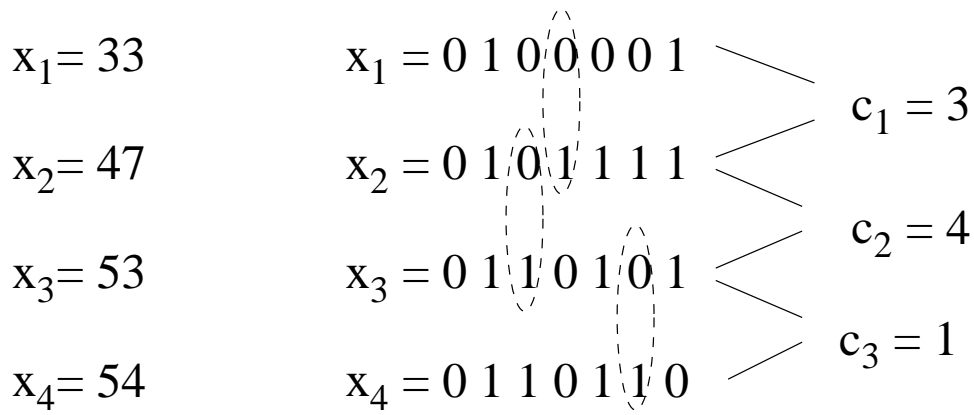


Figure 5: Calculation of c_i 's.

Proof: We prove this by contradiction. Let $c_j = c_h = k$ for some $1 \leq h \leq k - 1$. Then it means that all the higher order bit positions, i.e., more significant than c_j are same for all the x_i 's.

Without loss of generality let $x_j < x_h$. Since the x_i 's are ordered, we must have that the k^{th} bit of x_j is 0 and the k^{th} bit of x_{j+1} is 1. Also, we must have that the k^{th} bit of either x_h or x_{h+1} is 0. But this gives us that $x_{j+1} > x_h$ or $x_{j+1} > x_{h+1}$, which is not possible. Hence, j is unique. ■

We can represent the x_i values using a compressed trie. The trie has k leaves, containing the values 1 to k in order from left to right. Each internal node in the trie is labelled by the MSB position on which the elements of S indexed by the leaves in its sub-tree disagree. The following diagram depicts the trie corresponding to the sample data presented above:

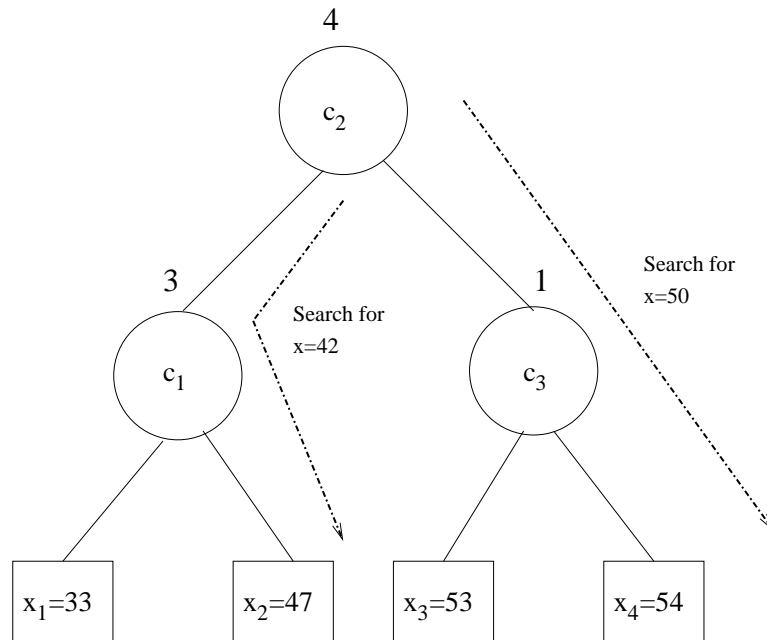


Figure 6: An example of a compressed trie for Q-heap.

The corresponding trie for S implements a form of decision tree that can be used to select a leaf

x_i for any value of $x \in U$. Starting at the root, branch left if the bit specified by the node's label is 0 and branch right if the bit specified by the node's label is 1.

We now proceed to give an algorithm for computing the $RANK()$ operation using Q-heaps. Specifically, the algorithm computes $rank_S(x)$, where we have $rank_S(x) = \#\{i | x_i \leq x \text{ for } x_i \in S\}$. If $x \in S$, then x leads to leaf x_i and $rank_S(x) = i$. Suppose $x \notin S$. The algorithm involves the following steps:

1. Find compressed trie T .
2. Find leaf x_i in T such that x leads to it.

Let d be the depth of leaf x_i , i.e., the number of internal nodes in T on the path from the root to leaf node x_i . Define $c_{j_0} = w > c_{j_1} > c_{j_2} \dots > c_{j_d} \geq 0$ such that $c_{j_1}, c_{j_2}, \dots, c_{j_d}$ are the labels in the path from the root to leaf x_i . Also, let $p = msb(x \oplus x_i)$. Here p represents the position of the most significant bit on which x and x_i differ. Note that $0 \leq p \leq w - 1$ since x and x_i are bit strings of length w and $x \neq x_i$.

3. Compute $rank_B(p)$, where $B = \{c_1, c_2, \dots, c_{k-1}\}$ and $c_1 < c_2 < \dots < c_{k-1}$

Let $q \in \{0, 1, \dots, d\}$ be such that $c_{j_q} > p > c_{j_{q+1}}$. Then q represents the first point in the uncompressed binary trie where x differs from x_i . Note that q is a function of T and $rank_B(p)$. Let v be the node on the path labelled by $c_{j_{q+1}}$ (or the leaf x_i if $q = d$).

4. Find relative values of x and x_i

If $x < x_i$, then x is smaller than all the elements that lead to v . Likewise, if $x > x_i$, then x is larger than all the elements that leads to v . From this, $rank_S(x)$ is obtained as follows:

$$rank_S(x) = \begin{cases} c_{j_{q+1}}, & \text{if } q = d & : \\ \min \text{ leaf in } v\text{'s subtree} - 1, & \text{if } x < x_i & : \\ \max \text{ leaf in } v\text{'s subtree}, & \text{if } x > x_i & : \end{cases}$$

For example,

$$x = 42 \text{ reaches } x_2 = 47, \text{ since } 42 < 47, \text{ so } rank(x) = 1$$

$$x = 48(0110000) \text{ reaches } x_3 = 53(0110101)$$

$$x = 50(0110010) \text{ reaches } x_4 = 54(0110110)$$

So, leaf reached in downward sweep is not necessarily closest to x .

Now, we need to show that each step in the above algorithm can be carried out in constant time. For this it is most critical to show that $rank_B(p)$ can be computed in $O(1)$ time.

We have $B = \{c_1, c_2, \dots, c_{k-1}\}$. Note that although set B do not contain duplicate elements. Let $r = |B|$ and let $b_0 < b_1 < \dots < b_{r-1}$ be the distinct values occurring in the sequence c_1, c_2, \dots, c_{k-1} . Define the bit string B' to be the concatenation of the binary representation of the elements of B with a 0 before each entry, i.e., $B' = 0b_00b_1 \dots 0b_{r-1}$. Note that $\lceil \log_2 w \rceil$ bits are sufficient to represent each element in B , since $b_i \in [0, w - 1], \forall b_i \in B$. Thus, the total number of bits in B' is equal to $r(\lceil \log_2 w \rceil + 1) \leq (k - 1)(\lceil \log_2 w \rceil + 1)$. Because $k < \frac{w}{\lceil \log_2 w \rceil} + 1$, B' fits in one word and all operations on B' take $O(1)$ time.

References

- [1] Michael L. Fredman, Dan E. Willard: Surpassing the Information Theoretic Bound with Fusion Trees. *J. Comput. Syst. Sci.* 47(3): 424-436 (1993).
- [2] Michael L. Fredman, Dan E. Willard: Trans-Dichotomous Algorithms for Minimum Spanning Trees and Shortest Paths. *J. Comput. Syst. Sci.* 48(3): 533-551 (1994).