

# 1 Constant-time Priority Queues

In this lecture, we will describe an implementation of a constant time priority queue [1]. In this implementation, both queries and updates take constant time in the worst case. We start with some definitions and prove some claims.

## 1.1 Definitions and Claims

**Definition 1** A branching node in a binary trie is a node with 2 children.

**Definition 2** A node  $v$  in a compressed binary trie is a right ancestor of a leaf  $x \in S$ , if  $x$  lies in the left sub-tree of  $v$ .

**Definition 3** A node  $v$  in a compressed binary trie is a left ancestor of a leaf  $x \in S$ , if  $x$  lies in the right sub-tree of  $v$ .

**Definition 4** The lowest right ancestor of a leaf  $x \in S$  is the lowest branching node that is a right ancestor of  $x$ . These are also well defined for  $x \in U - S$ , where ancestor is defined as a branching node that has  $x$  in the left sub-tree in the OR-tree representation.

**Definition 5** The lowest left ancestor of a leaf  $x \in S$  is the lowest branching node that is a left ancestor of  $x$ . These are also well defined for  $x \in U - S$ , where ancestor is defined as a branching node that has  $x$  in the right sub-tree in the OR-tree representation.

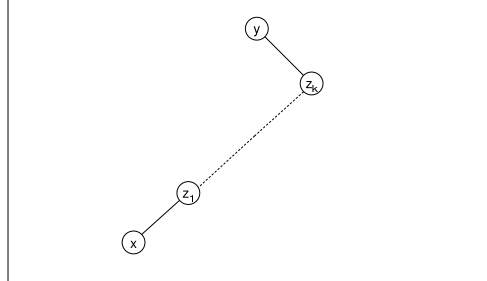
Homework 1, Problem 1 :-

Consider a priority queue storing the set  $S$  using a compressed bit-vector OR-trie. Define left, right, lowest left and lowest right ancestors uniformly for all  $x \in U$ , in terms of the labels on the nodes in the compressed binary trie.

We will now prove a few claims that will help in analyzing the complexity of queries and updates to Priority Queues.

**Claim 1** *If  $x \in S$ , then*

- *lowest left ancestor of  $x$  points to  $x$*
- *lowest right ancestor of  $x$  points to  $x$*



**Proof:** Let  $y$  be the lowest left ancestor of  $x$ . Then the path  $x, z_1, \dots, z_k, y$  in the tree is as follows: So,  $x$  is in the left subtree of its ancestors  $z_1, \dots, z_k$  but it is in the right subtree of  $y$ . Therefore,  $x$  is the minimum in  $y$ 's subtree, so there is a pointer from  $y$  to  $x$ .

Now, let  $y$  be the lowest right ancestor of  $x$ . Then by a similar argument,  $x$  is the largest leaf in  $y$ 's left subtree, so there is a pointer from  $y$  to  $x$ . ■

**Claim 2** *If  $x \in S$ , then exactly 2 pointers point to  $x$ .*

**Proof:** Suppose  $y$  points to  $x$ . Then, either  $x$  is the smallest leaf in  $y$ 's right subtree, or the largest leaf in  $y$ 's left subtree. In the first case,  $y$  is the lowest left ancestor of  $x$ , and in the second case,  $y$  is the lowest right ancestor of  $x$ . So, the only (internal) pointers to  $x$  are from the lowest left and lowest right ancestors of  $x$ .

There are 3 cases for  $x$ .

$x$  is the minimum in  $S$

Here  $x$  has an external pointer pointing to it as  $\min(S)$ . Also  $x$  does not have a left ancestor, while its lowest right ancestor is its parent. So,  $x$  has only one internal pointer pointing to it, giving exactly 2 pointers.

$x$  is the maximum in  $S$

Here  $x$  has an external pointer pointing to it as  $\max(S)$ . Also  $x$  does not have a right ancestor, while its lowest left ancestor is its parent. So  $x$  has only one internal pointer to it, giving exactly 2 pointers.

$x$  is neither the minimum or maximum of  $S$

Here  $x$  has both the lowest left ancestor and the lowest right ancestor, so  $x$  has 2 pointers pointing to it. It has no external pointers. So, it has a total of 2 pointers pointing to it. ■

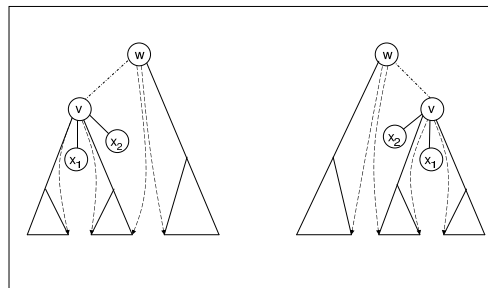
**Claim 3** If  $x \in U - S$ , then either

- the lowest left ancestor of  $x$  points to  $Pred(x)$  and  $Succ(x)$ , or
- the lowest right ancestor of  $x$  points to  $Pred(x)$  and  $Succ(x)$

**Proof:** Assigned as Homework 1, Problem 2.

Sketch of Proof:-

Consider the 2 cases as given in the figure below, where  $x$  lies in  $w$ 's left (right) subtree.  $x$  can now lie in 2 positions w.r.t. the compressed binary trie, i.e.  $x_1$  or  $x_2$ . Now  $v$  is the lowest left (right) ancestor of  $x_1$  and points to  $Pred(x)$  and  $Succ(x)$ . Alternatively,  $w$  is the lowest right (left) ancestor of  $x_2$  and points to  $Pred(x)$  and  $Succ(x)$ . Complete the proof on these lines. ■



From these claims, we can see that to perform  $Pred(x)$  and  $Succ(x)$ , it is sufficient to find the lowest left and the lowest right ancestor of  $x$ .

Note:- Each node in a compressed binary trie has pointers to largest leaf in its left subtree and the smallest leaf in its right subtree. There are also pointers to minimum and maximum leaves. In addition, we have successor and predecessor pointers at each of the leaves.

## 1.2 Operations

Find - min(x)

Follow min pointer  $\rightarrow$  takes  $O(1)$

Pred(S, x), Succ(S, x)

Find lowest left ancestor and lowest right ancestor of  $x$ , by following a path from root to a leaf, and keeping track of the last left ancestor and last right ancestor seen. It takes  $O(\log n)$  time. Can we do this faster?

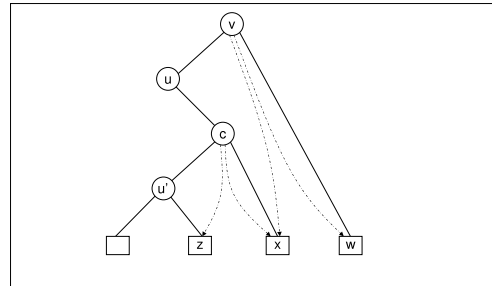
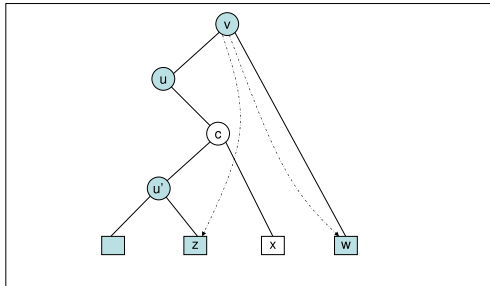
Note:- If  $x \notin S$ , then path ends at the lowest ancestor.

If  $x \in S$ , maximum of left tree pointer of lowest left ancestor points to  $Pred(x)$ , and minimum of right tree pointer of lowest right ancestor points to  $Succ(x)$ . If  $x \notin S$ , then either lowest left ancestor or lowest right ancestor points to  $Pred(x)$  and  $Succ(x)$ , so check all 4 leaves.

Now, if finding left/right ancestor is  $O(n)$ , then  $Pred(x)$  and  $Succ(x)$  are also  $O(n)$ .

Insert( $S, x$ )

- Find lowest right ancestor  $v$  of  $x$
- Follow pointer from  $v$  to largest leaf  $z$  in  $v$ 's left subtree
- Determine lowest common ancestor  $c$  of  $x$  and  $z$  in the compressed binary tree ( $c$  is not necessarily a branching node).  $z, x, c$  are all in  $v$ 's left subtree.



Case 1:  $z < x$

**Claim 4** *If  $z < x$ , then right subtree of  $c$  is empty.*

**Proof:** If there is a leaf  $y \in S$  in  $c$ 's right subtree, then  $z > y$ . This contradicts the fact that  $z$  is the largest leaf in  $v$ 's left subtree, since  $y$  is also in  $v$ 's left subtree. ■

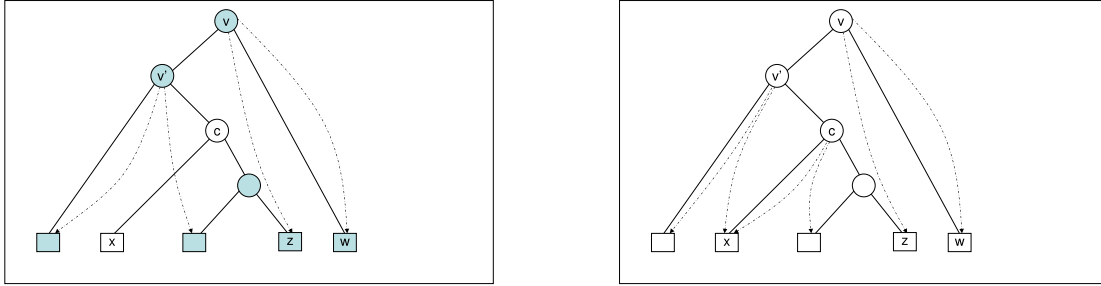
This proves that  $c$  is not a node in the compressed binary trie.

- Insert  $c$  as a right child of node  $u$  which is the longest prefix of  $c$
- right child  $u'$  of  $u$  is now  $c$ 's left child
- $x$  is now  $c$ 's right child
- set  $c$ 's pointers to  $z$  and  $x$
- replace  $v$ 's left tree pointer from  $z$  to  $x$

Case 2:  $z > x$

**Claim 5** *If  $z > x$ , then left subtree of  $c$  is empty.*

**Proof:** If there is a leaf  $y \in S$ , in  $c$ 's left subtree, then  $z \in S$  is in  $c$ 's right subtree,  $c$  must be a branching node. Since  $z$  is in  $v$ 's left subtree, it follows that  $c \neq v$ , so  $v$  is a proper ancestor of  $c$ . Since  $c$  is a right ancestor of  $x$ , but is lower than  $v$ , this contradicts the fact that  $v$  is the lowest right ancestor of  $x$ . ■



Let  $v'$  be the lowest left ancestor of  $x$ . Either  $v$  is ancestor of  $v'$  or  $v'$  is ancestor of  $v$ . In either case, either  $v$  or  $v'$  points to  $Succ(x)$ .

Move  $v$  or  $v'$  pointer from  $Succ(x)$  to  $x$ . Set  $c$ 's pointers to  $x$  and  $Succ(x)$ .

Delete – Min( $x$ )

- Set min to  $Succ(x)$
- Remove lowest common ancestor of  $x$  and  $Succ(x)$  in the compressed binary trie (parent of  $x$ )
- No pointers need to be changed
- This takes  $O(\log m)$  time

We thus see that all operations would be  $O(1)$ , if lowest ancestors can be computed in  $O(1)$  time. We show that this can be done using the RAMBO model, described in the next section.

## 2 RAMBO Model

The RAMBO model, introduced by Fredman and Saks [2], stands for *Random Access Machine with Bit Overlap*. It has been implemented in real hardware using SDRAM.

Idea :- Individual Bits appear simultaneously in multiple words.

An Example

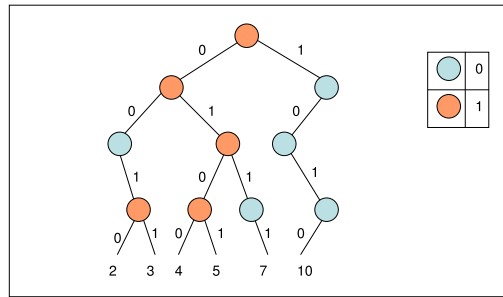
Consider an  $n \times n$  matrix,

$$\begin{bmatrix} a_{11} & a_{12} & \dots & a_{1n} \\ a_{21} & a_{22} & \dots & a_{2n} \\ \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot \\ a_{n1} & a_{n2} & \dots & a_{nn} \end{bmatrix}$$

Let  $R[i]$  store the  $i^{th}$  row  $a_{i1}, a_{i2}, \dots, a_{in}$

Let  $C[i]$  store the  $i^{th}$  column  $a_{1i}, a_{2i}, \dots, a_{ni}$

We see that the matrix is represented using  $2n$  words. Thus, each bit in the matrix occurs as part of 2 words. Conversely, every  $R[i]$  and  $C[j]$  pair have one bit in common  $a_{ij}$



We implement a priority queue using the RAMBO model as follows:

A log  $m$ -bit string represents each path from root to the parent of a leaf in the uncompressed binary trie. Each node on the path has value 1, if it is a branching node, and 0 otherwise.

Now,  $R[i]$  stores the bit string indicating which nodes on the path from the root to leaf  $(2i - 1)$  and  $(2i)$  are branching nodes.

$$R[1] = 1101 \text{ (the path to 2 and 3)}$$

$$R[3] = 1110 \text{ (the path to 6 and 7)}$$

Note :-  $R[1]$  and  $R[3]$  share 2 bits.

The path to leaf  $x$  is stored in  $R[\lfloor \frac{x}{2} \rfloor]$



The only right ancestor is at position 4, from the right. Hence, it is also the lowest right ancestor. Now shift  $\text{bin}(x)$  to the right by 4 giving  $\varepsilon$  (the root which is the lowest right ancestor).

Note :- The lowest left (right) ancestor is the leftmost (rightmost) 1-bit in the resulting string.

This gives us a constant time algorithm for finding the lowest left ancestor and lowest right ancestor, by doing  $O(1)$  time binary operations on binary words of length  $O(\log m)$ . We assume arithmetic operations on words are constant time.

## 4 Finding Lowest Common Ancestor

Finding lowest common prefix of  $\text{bin}(x)$  and  $\text{bin}(z)$  gives us the lowest common ancestor of  $x$  and  $z$ .

Perform,  $\text{bin}(x) \oplus \text{bin}(z)$  and find the leftmost 1.

The location of the leftmost 1 is first bit in which  $x$  and  $z$  vary.

e.g.  $x = 10$   $\text{bin}(x) = 1010$ ,  $z = 9$   $\text{bin}(z) = 1001$ ,  $x \oplus z = 0011$

The leftmost 1 occurs at bit position 2. Now, shift  $x$  right by 2 giving 10, the lowest common ancestor of  $x$  and  $z$ .

## 5 Finding leftmost or rightmost 1-bit in a String

This is needed in both the preceding algorithms.

Solution 1:- There may be a built-in operation in some computers, e.g. Cray, Pentium.

Solution 2:- Perform integer to floating point conversion.

To find leftmost (most significant) 1-bit

A floating point number consists of a fraction  $r$ ,  $2^{-1} \leq r < 2^0$  and an exponent  $k$ ; representing the number  $r \times 2^k$ . We convert a string  $x_{n-1} \dots x_0$  from integer to floating point notation, and the exponent field gives us the location of the leftmost (most significant) 1-bit.

e.g.  $0010100 = .101 \times 2^5 \rightarrow$  leftmost 1-bit is  $5^{th}$  MSB.

Now, how is the floating point number represented?

4 bits to represent fraction, followed by 3 bits to represent exponent.

e.g.  $.101 \times 2^5 \rightarrow 1010101$

Hence, to get the exponent, take bitwise AND with 0000111.

e.g.  $1010101 \wedge 0000111 = 0000101 \rightarrow 5$ .

This gives the most significant 1-bit position, in constant time.

To find rightmost (least significant) 1-bit

First subtract 1 from  $w$ .

Note that when we subtract 1, we negate all bits starting at the least significant bit upto the least significant 1-bit. All other bits remain the same.

e.g.  $w = 101101000$ , so  $(w - 1) = 101100111$

Take the bitwise AND of  $w$  and  $w - 1$ .

This differs from  $w$  only in 1 bit. The rightmost 1-bit in  $w$  becomes a 0-bit in  $w \wedge (w - 1)$ .

e.g.  $w \wedge (w - 1) = 101100000$

Take the bitwise XOR of  $w$  and  $w \wedge (w - 1)$ .

This results in a string of all zeroes, and exactly one 1-bit, at the location of the rightmost 1-bit in  $w$ .

e.g.  $w \oplus w \wedge (w - 1) = 000001000$

This gives the least significant 1-bit position, in constant time.

**Solution 3**:- Use a Lookup Table.

We can use a lookup table that maps every string to a pre-determined value indicating the position of the leftmost/ rightmost 1-bit. We would require a table of  $m$  words. The bit positions

can be stored in  $O(\log \log m)$  bits and hence we would require  $O(m \log \log m)$  space.

Alternatively, we can use a table that maps strings of half the size, i.e. lookup table stores  $\sqrt{m}$  words of length  $\frac{\log m}{2}$  each. (Note that there are  $2^{\frac{\log m}{2}} = \sqrt{m}$  words of  $\frac{\log m}{2}$  length).

e.g. Consider a string  $u = u_1 u_2$  where  $|u_1| = |u_2|$

To search for the leftmost 1-bit in  $u$ ,

- test if  $u_1 = 0$
- if so, find leftmost 1-bit in  $u_2$
- if not, find leftmost 1-bit in  $u_1$

To search for the rightmost 1-bit in  $u$ ,

- test if  $u_2 = 0$
- if so, find rightmost 1-bit in  $u_1$
- if not, find rightmost 1-bit in  $u_2$

This gives a constant time algorithm to find the leftmost and rightmost 1-bit positions in a string. The table has  $\sqrt{m}$  entries, each taking  $\log\lceil\frac{\log m}{2}\rceil = O(\log \log m)$  space. Hence, the total space required is  $O(\sqrt{m} \log \log m)$ .

Thus, given a fixed universe, we have sub-logarithmic time algorithms for queries and updates to the priority queue. Now let us assume that the universe is not fixed.

Consider the *decision-tree model*. The algorithm is viewed as a tree, where each node represents computation and branching is represented by  $O(1)$  edges outgoing from the node. The length of a root-to-leaf path is the running time of a particular instance of the algorithm.

Note:- It does not model random access since this allows multi-way branching.

## 6 Lower Bound of Sorting Algorithms

**Lemma 6** *Any comparison based sorting algorithm takes  $\Omega(n \log n)$  time.*

**Proof:** A decision tree for sorting has at least  $n!$  leaves since there are  $n!$  possible permutations for a set of  $n$  values. Thus the height of the tree is at least  $\log(n!) \approx (n \log n)$ . ■

We now prove by reduction that any data structure implementing insert and successor operations requires  $\Omega(\log n)$  amortized time in the worst case, in the decision tree model.

**Lemma 7** *For any data structure that implements insert and successor operations, a sequence of  $\Theta(n)$  insert and successor operations, take  $\Omega(n \log n)$  time in the worst case.*

**Proof:** Let us assume that the lemma is not true. Then we construct an implementation for sorting which takes  $o(n \log n)$  time, giving us a contradiction.

To sort  $x_1, x_2, \dots, x_n$

  insert ( $x_1$ )

  insert ( $x_2$ )

  .

  .

  insert ( $x_n$ )

$y_1 = \text{successor}(-\infty)$  or  $y_1 = \text{min}(\text{Data Structure})$

$y_2 = \text{successor}(y_1)$

  .

  .

$y_n = \text{successor}(y_{n-1})$

  return ( $y_1, y_2, \dots, y_n$ )

Since the sequence of  $2 \times n$  operations takes  $o(n \log n)$  time, the sorting algorithm takes  $o(n \log n)$  to return the sorted list  $y_1, y_2, \dots, y_n$ , which is a contradiction. ■

**Theorem 8** *For any data structure that implements insert and successor operations, the amortized time complexity of each operation is  $\Omega(\log n)$*

**Proof:** Since a sequence of  $\Theta(n)$  operations takes  $\Omega(n \log n)$  time in the worst case, it follows that the amortized time complexity of each operation is  $\Omega(\log n)$ . ■

## 7 B-Trees

The next lecture would talk about obtaining constant time operations for priority queues based on B-Trees.

## Review of B-Trees:-

- A B-tree is a balanced search tree.
- each internal node stores between  $(t-1)$  and  $(2t-1)$  keys  
(Hence, minimum degree =  $t$  and maximum degree =  $2t$ ).
- node  $x$  stores  $k$  keys  $v_1, v_2, \dots, v_k$  where  $v_1 < v_2 < \dots < v_k$
- node  $x$  has  $k+1$  children  $y_1, y_2, \dots, y_{k+1}$ 
  - leaves in subtree rooted at  $y_1$ , have keys  $v \leq v_1$
  - leaves in subtree rooted at  $y_i$ ,  $1 \leq i \leq k$  have keys  $v_{i-1} < v \leq v_i$
  - leaves in subtree rooted at  $y_{k+1}$  have keys  $v > v_k$

## References

- [1] Andrej Brodnik, Svante Carlsson, Johan Karlsson, J. Ian Munro, *Worst case constant time Priority Queue*, SODA 2001: 523-528
- [2] Michael L. Fredman, Michael E. Saks, *The Cell Probe Complexity of Dynamic Data Structures*, STOC 1989: 345-354