

TOPICS COVERED

1. Priority Queues
2. Comparison based Data Structures
3. Integer Data Structures
4. Dynamic Graph Data Structures
5. Cache-oblivious Data Structures
6. Distributed Data Structures
7. Randomized Data Structures

COURSE REQUIREMENTS

1. Scribe notes
2. 2-3 Homework assignments
3. Research project (paper and presentation)

Class Time: WF 2:10-3:25

1 Introduction

Abstract Data Type (ADT): A collection of mathematical objects and operations on these objects. Eg. a priority queue.

Objects: Objects are subsets of an ordered universe, U .

Operations: Operations are either queries or updates. Queries do not change the object. An operation is called an update if it changes the object. For example, $\text{Find_Min}(S)$ – Query

Insert(S,x) – Update

Delete_Min(S,x) – Update.

A static ADT has only query operations. A dynamic ADT has both query and update operations. We will study dynamic ADTs in this course. A few examples of static ADTs are as follows:

1. Static Predecessor ADT:

- Subsets of an ordered universe, U
- operation $Pred(S, x), S \subseteq U, x \in U$
- Returns $\max\{y | y \in S, y < x\}$, and \perp if no such y exists.

2. Static Dictionary:

- Subsets of an unordered universe U
- Operation $Member(S, x), S \subseteq U, x \in U$
- Returns true iff $x \in S$.

Note: Dynamic Dictionary is a dynamic ADT with additional operations INSERT(S,x), DELETE(S,x).

A data structure is an implementation of an ADT. It includes the representation of the object as well as algorithms for the operations. Next, consider the Priority queue ADT for a universe, U , of size m , where $U = \{0, \dots, m - 1\}$. The set size $|S| = n$. Table 1 lists complexities for some priority queue data structures.

Table 1: Complexities of various operations

DS type	Find_Min(S)	Insert(S,x)	Delete_Min(S)	Pred(S,x)	Space
Unordered list	$\theta(n)$	$\theta(1)$	$\theta(n)$	$\theta(n)$	$\theta(n)$
Ordered list	$\theta(1)$	$\theta(\log n)$	$\theta(\log 1)$	$\theta(n)$	$\theta(n)$
Heap	$\theta(1)$	$\theta(\log n)$	$\theta(\log n)$	$\theta(n)$	$\theta(n)$
Balanced binary search tree	$\theta(\log n)$	$\theta(\log n)$	$\theta(\log n)$	$\theta(\log n)$	$\theta(n)$
Bit vector	$\theta(m)$	$\theta(1)$	$\theta(m)$	$\theta(m)$	$\theta(m)$
Bit vector with OR-tree	$\theta(\log m)$	$\theta(\log m)$	$\theta(\log m)$	$\theta(\log m)$	$\theta(m)$
Compressed binary trie	$\theta(\log m)$	$\theta(\log m)$	$\theta(\log m)$	$\theta(\log m)$	$\theta(n)$

Fact 1 $Member(S,x)$ takes $\Omega(n)$ time in an unordered list (simply because every element has to be looked at).

Claim 2 $Member(S,x)$ takes $\Omega(n)$ time in a heap.

Proof: *By Reduction method:* We will first prove it by reduction from search in an unordered list. Suppose, for Contradiction, there is an $o(n)$ algorithm for $\text{Member}(S,x)$ in a heap. We construct a $o(n)$ algorithm for $\text{Member}(A,x)$ in an unordered list A .

Consider a heap S where all elements of A are at the bottom level and every other element is $-\infty$. The size of the heap is $\Theta(n)$. Now $\text{Member}(S,x)$ iff $\text{Member}(A,x)$. Compute $\text{Member}(S,x)$ in $\Theta(n)$ time. Since x is a member of S iff x is a member of A , this gives a $o(n)$ algorithm for $\text{Member}(A,x)$ which is a contradiction by Fact 1. ■

Proof: *By Adversary Argument:* IDEA: Adversary changes the input based on steps of the algorithm to produce the wrong result. A sketch of the proof for the claim using adversary argument is as follows:

Suppose, algorithm A exists for $\text{Member}(S,x)$ that runs in $o(n)$ time. Let x be an element not in S , where $x > y$, for all $y \in S$. Clearly, $\text{Member}(S,x)$ returns nil. Since it runs in $o(n)$ time, it could not have observed all the $\lceil \frac{n}{2} \rceil$ leaves. Let $S[i]$ be a leaf not scanned, and let S' be a new heap where $S'[i] = x$ and $S'[j] = S[j]$, for all $j \neq i$. Now, $\text{Member}(S',x)$ will also return nil. ■

Next, we study some priority queue data structures.

2 Bit vectors

A bit vector is an array $B[0, \dots, m-1]$ such that

$$B[i] = \begin{cases} 1, & i \in S \\ 0, & i \notin S \end{cases}$$

$\text{Insert}(S,x)$ can be implemented in $\Theta(1)$ time by setting $B[x] = 1$. Find_Min and Delete_Min require $\Theta(m)$ time since in the worst-case, every bit is queried. The space required is clearly $\Theta(m)$. $\text{Pred}(S,x)$ can be implemented in $\Theta(m)$ time by querying $B[x-1], B[x-2], \dots$ until a 1-bit is found.

3 Bit vector with OR-tree

We add a binary tree T whose leaves are the elements of the vector in order. Each node v in the tree has a value $\text{val}[v]$. For leaves, if $v = B[x]$, then $\text{val}[v] = 1$ iff $x \in S$. For internal nodes, $\text{val}[v] = \text{val}[\text{left}(v)] \vee \text{val}[\text{right}(v)]$. See Figure 1.

The operations are implemented as follows:

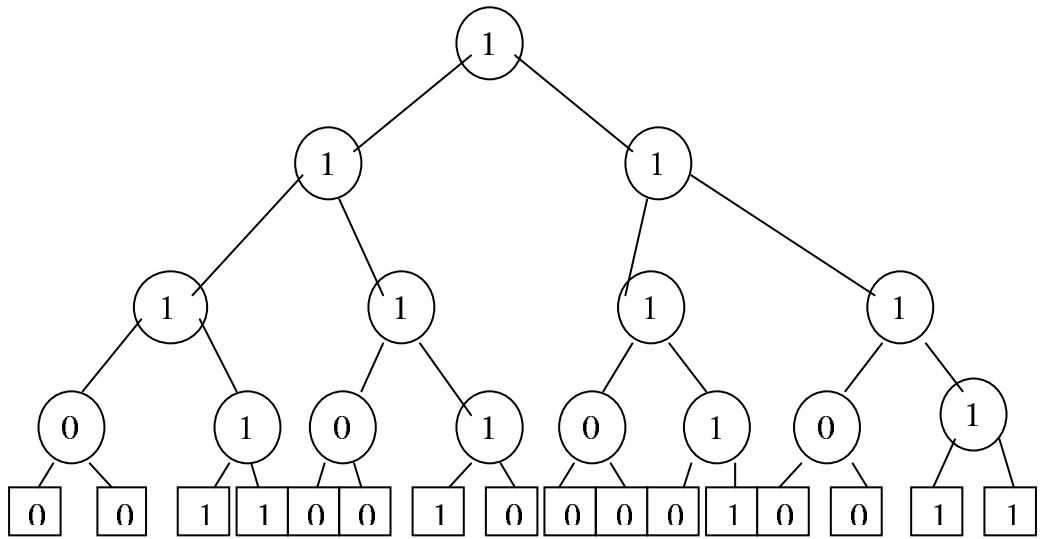


Figure 1: Bit vector with OR-tree

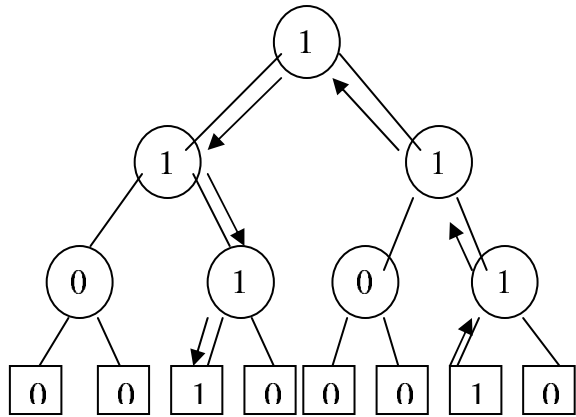


Figure 2: Pred(T,x)

Find_Min(T)

1. if $val[root(t)] = 0$ then return \perp
2. $v \leftarrow root[T]$
3. While v is not a leaf
4. if $val[left[v]] = 1$
5. then $v \leftarrow left[v]$
6. else $v \leftarrow right[v]$
7. return $index[v]$

Note: For a Leaf $v = B[x]$, we have $index[v] = x$.

The algorithm repeatedly follows the left-most child whose value is 1, starting at the root. If $root[T]$ has value 0 then the priority queue is empty, so the algorithm returns \perp .

Insert(T,x)

1. $v \leftarrow B[x]$
2. do
3. $val[v] \leftarrow 1$
4. $v \leftarrow parent[v]$
5. until $v = root[T]$
6. $val[root[T]] \leftarrow 1$

Procedure $Insert(T,x)$ sets the value of leaf $B[x]$ and ancestors to 1.

Delete_Min(T)

1. if $val[root[T]] = 0$ then return
2. $v \leftarrow B[Find_Min[T]]$
3. $val[v] \leftarrow 0$
4. while $v \neq root[T]$ and $val[sibling(v)] = 0$, do
5. $val[parent[v]] \leftarrow 0$

6. $v \leftarrow \text{parent}[v]$

Delete_Min finds the minimum element x using Find_Min(T). It sets the value of $B[x]$ to 0, and then modifies the value of its ancestors to preserve OR.

Pred(T,x)

1. $v \leftarrow B[x]$
2. while $v \neq \text{root}[T]$ and not ($v = \text{right}[\text{parent}(v)]$ and $\text{val}[\text{left}[\text{parent}(v)]] = 1$) do
3. $v \leftarrow \text{parent}[v]$
4. if $v = \text{root}[T]$ then return \perp
5. $w \leftarrow \text{left}[\text{parent}[v]]$
6. while w is not a leaf do
7. if $\text{val}[\text{right}[w]] = 1$
8. then $w \leftarrow \text{right}[w]$
9. else $w \leftarrow \text{left}[w]$
10. return $\text{index}[w]$

Pred(T,x) is implemented by following parent pointers from $B[x]$ until a node v is entered from the right and the value of $\text{left}[v]$ is 1. The algorithm returns the maximum leaf in the sub-tree rooted at $\text{left}[v]$, by repeatedly following the rightmost child whose value is 1. See Figure 2.

Find_Min, Insert, Delete_Min and Pred all take $\Theta(\log m)$ time. Space required is $\Theta(m)$.

Consider modifying the data structure as follows:

1. Each leaf v with $\text{val}[v] = 1$ has a pointer $\text{pred}[v]$ to the predecessor of v and a pointer $\text{succ}[v]$ to the successor of v .
2. Label each left branch with 0 and right branch with 1. We identify every node v in the tree with the binary string obtained by following the path from the root to v .

Pred(T,x)

1. Find the longest prefix y of x whose value is 1
2. If $y = x$, then,

3. x is in the priority queue, so return $index[pred[x]]$
4. If y is an internal node of T , then,
 5. if $x = y0x'$, then, (See Figure 3)
 6. $y1$ has value 1 and $y0$ has value 0. The sub-tree rooted at $y0$ contains only 0-valued leaves, so find the smallest 1-valued leaf in the sub-tree rooted at $y1$. Now, $pred[w]$ is the predecessor of x , since w is the successor of x .
 7. else if $x = y1x'$, then,
 8. $y0$ has value 1 and $y1$ has value 0. The sub-tree rooted at $y1$ has only 0-valued leaves so largest 1-valued leaf in sub-tree rooted at $y0$ is the predecessor of x .

This algorithm takes $\Theta(\log m)$ time.

Delete_Min is as before except that the successor of the deleted node must have its predecessor pointer updated. Similarly, predecessor of deleted node must have its successor pointer updated.

Insert: The successor and predecessor of the inserted node have to be found and their pointers updated.

The operations take $\Theta(\log m)$ time and space is still $\Theta(m)$.

4 Compressed Binary Trie

We look at modifications that will reduce space required, without affecting runtime of operations:

1. Replace 0-valued nodes with NIL, since sub-trees rooted at these nodes contain only 0-valued nodes. Now, remaining nodes all have value 1, so there is no need to store these values. This forms a binary trie. See Figure 4 and Figure 5.

Paths in a binary trie are determined by the data stored (efficient storage). In our implementation we identify each node v with the binary string representing the path from the root to v .

The binary trie has space $\Theta(n \log m)$ where $m = |U|$ and $n = |S|$. The trie has height $\log m$ and there are $O(n)$ nodes at each level. Actually, space = $\Theta(\min(n \log m, m))$.

2. Now, we "compress" the tree by only representing the internal nodes with two non-NIL children, i.e we remove internal nodes with one child.⁴ We label each edge (u, v) in the compressed trie with the sub-string representing the path from u to v in the binary trie. This produces a compressed binary trie. See Figure 6.

The compressed binary trie has $\Theta(n)$ nodes. Since, there are n leaves and the number of internal nodes is equal to the number of leaves minus 1, so there are $n - 1$ internal nodes.

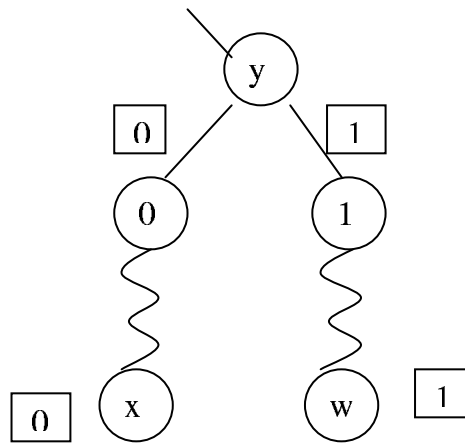


Figure 3: Pred(T,x)

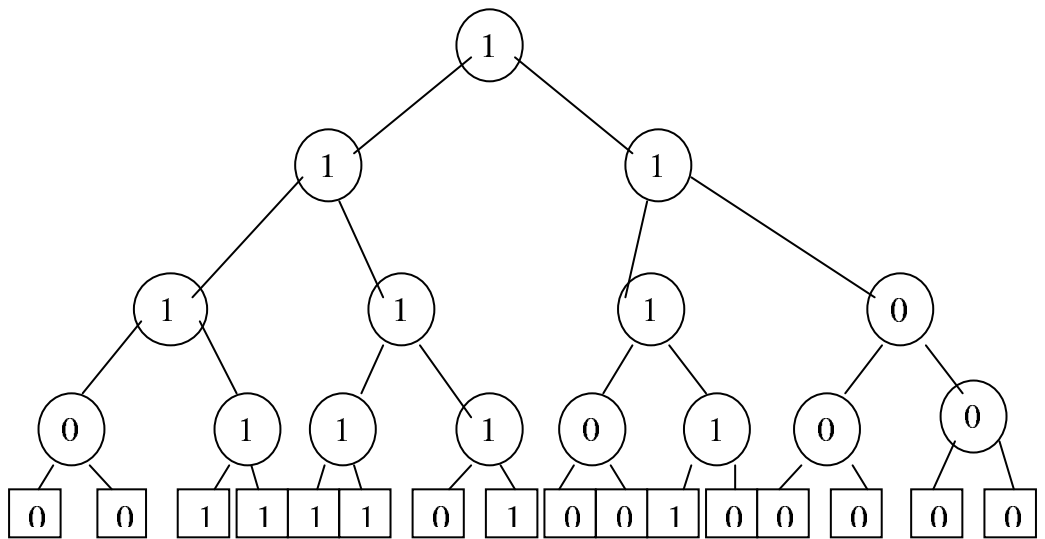


Figure 4: Bit Vector OR-Tree

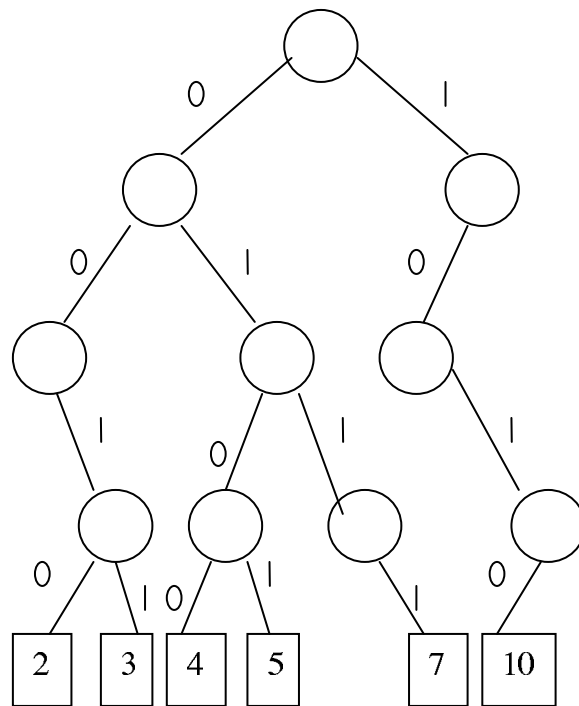
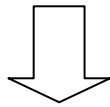


Figure 5: Binary Trie

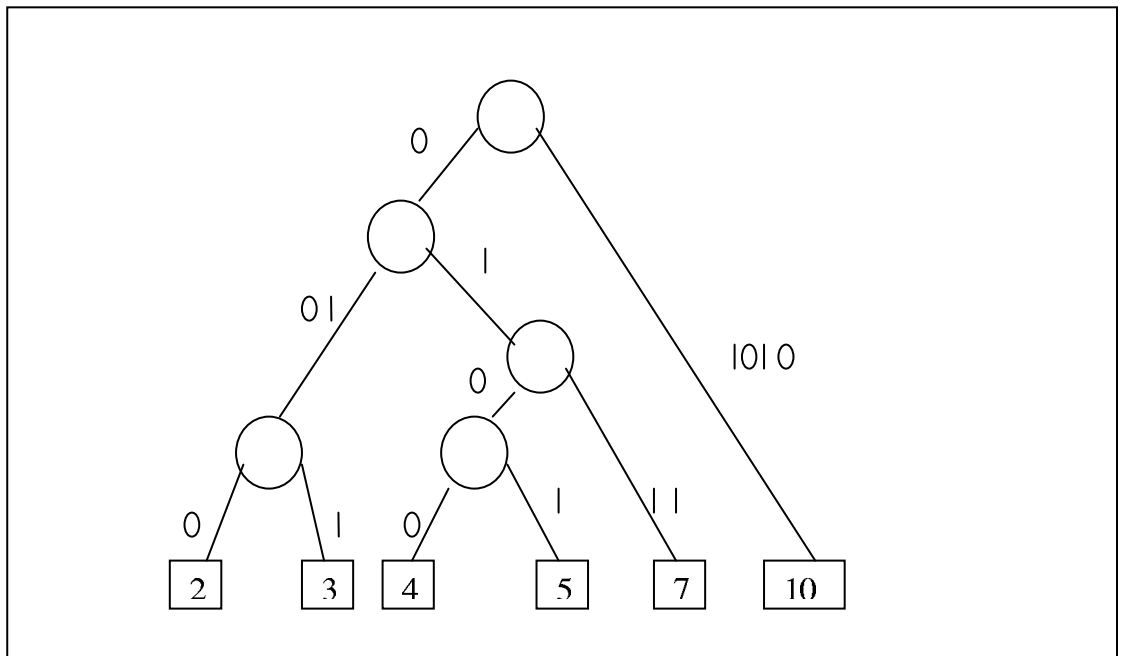


Figure 6: Compressed Binary Trie

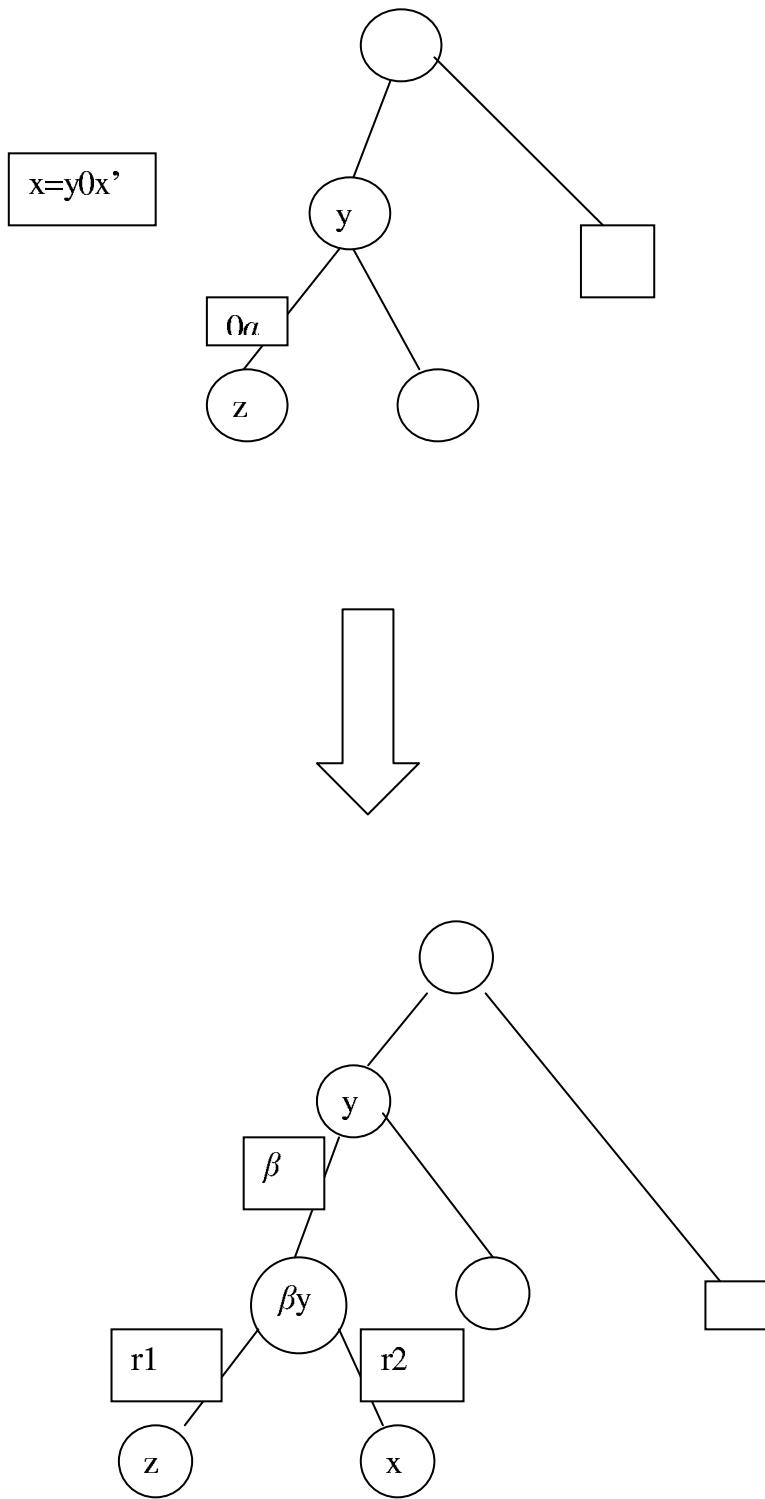


Figure 7

The priority queue operations are modified as follows:

Find_Min(T)

Repeatedly follow leftmost child starting from the root (earlier we followed the leftmost 1-valued child, but now all children are 1-valued).

Insert(T,x)

1. Find the node y that is the longest prefix of x in T .
2. If $x = y$, then x already exists, so we are done.
3. if $x = y0x'$, (see Figure 7)
4. Let z be the left child of y i.e. $z = y0\alpha$, and (y, z) is labeled with 0α .
5. Let β be the longest prefix of 0α that is also a prefix of $0x'$. Now $1 \leq |\beta| \leq 1 + |\alpha|$.
6. Add a new leaf x and a new internal node $y\beta$.
7. Node $y\beta$ is the left child of y and its 2 children are z and x .
8. Edge $(y, y\beta)$ is labeled with β .
9. Edge $(y\beta, z)$ is labeled with r_1 where $z = y\beta r_1$.
10. Edge $(y\beta, x)$ is labeled with r_2 where $x = y\beta r_2$.
11. If $x = y1x'$,
12. Same as above, except replace left with right.

Example: Insert(T,13). See Figure 8 and Figure 9.

$x = 1101, y = 1, \Rightarrow x = y101$
 $z = y110$ (right child of y)
 $\Rightarrow \beta = 1$ (longest common prefix of 101 and 110),
so add node $y\beta = 11$.

The left child is $x = 1101$ and the right child is $z = 1110$.

Since the height of the tree is $\Theta(\log m)$, not $\Theta(\log n)$, this takes $\Theta(\log m)$ time.

Delete_Min(T)

1. Find the minimum element x using Find_Min(T).
2. Let y be the parent of x , w the sibling of x , and z the parent of y .

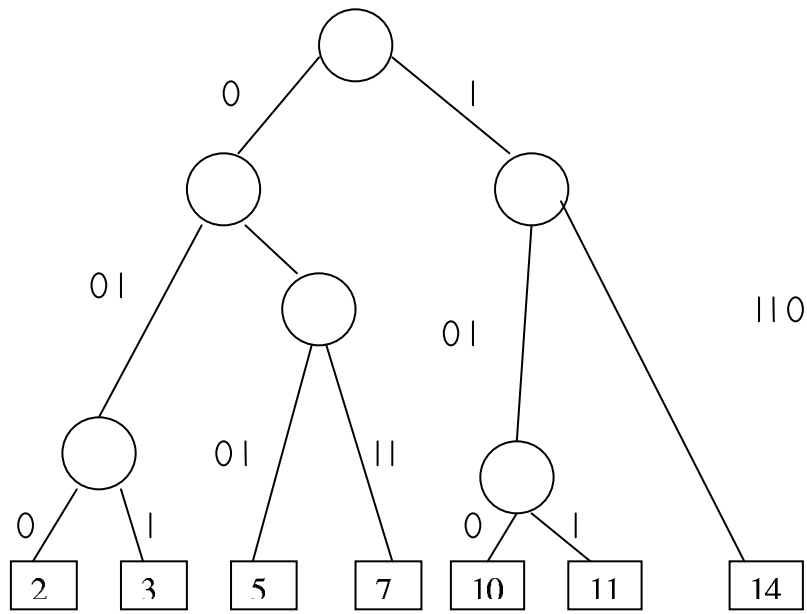


Figure 8

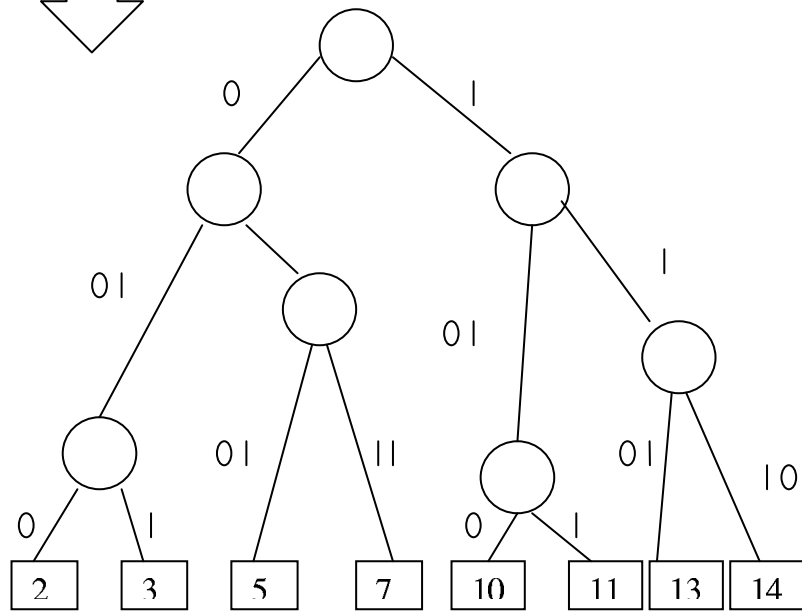
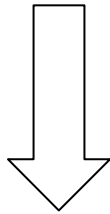


Figure 9

3. Delete y and x and let z be the new parent of w .
4. Remove edges (z, y) , (y, x) and (y, w) and add (z, w) whose label is $label(z, y) \cdot label(y, z)$.

This takes $\Theta(\log m)$ time.

Pred(T,x): This is the same as before.

So, $O(\log m)$ is the worst case for all operations. Can this be improved? We describe a structure defined by van Emde Boas et al [1].

Goal: Achieving $O(\log \log m)$ time for $\text{Pred}(T,x)$.

Let $h = \lceil \log m \rceil$, be the height of the tree. Note that balanced binary search trees get $O(\log n)$ performance, but not sub-logarithmic. We are able to do better here because we have a fixed universe. The universe is not the set of all integers, but of finite size m . i.e. $U = \{0, 1, \dots, m-1\}$.

Idea: We need to compute the longest prefix of x that is a node in the trie. Using binary search takes time that is $O(h)$. How do we get $O(\log h)$ complexity? We do binary search over $O(h)$ items. To do this, we build the tree in such a way that at each search step we can reduce the height of the tree to be searched by half.

Consider the non-compressed binary trie:

- Searching for node x , instead of going to b , where $x = bx''$, $b \in \{0, 1\}$, we go to x' , where $x = x'x''$, $|x'| = \lceil \frac{h}{2} \rceil$.
- Store "middle nodes" x' if the binary trie in a dictionary.
- Store pointers at every node x' to the minimum node $\min(x')$ and the maximum node $\max(x')$ in subtree rooted at x' .

Let

$$S' = \{x' \mid \exists x'', x'x'' \in S, |x'| = \lceil \frac{|x|}{2} \rceil, |x''| = \lfloor \frac{|x|}{2} \rfloor\}$$

Also,

$$S_{x'} = \{x'' \mid x'x'' \in S, |x'| = \lceil \frac{|x|}{2} \rceil, |x''| = \lfloor \frac{|x|}{2} \rfloor\}$$

Given tree S , we construct the tree S' , and for each x' , the tree $S_{x'}$. Each of these is a tree whose paths define strings of half the length of the paths defined in S . So, S' is the tree consisting of the top $h/2$ levels of the tree S . Each tree $S_{x'}$ is a subtree of S rooted at a node at level $h/2$.

This structure is defined recursively. See Figure 10. Each of the sub-trees in the figure is further subdivided in a similar fashion. Therefore, after each step in the search, we are able to reduce the height of the subtree to be searched by half.

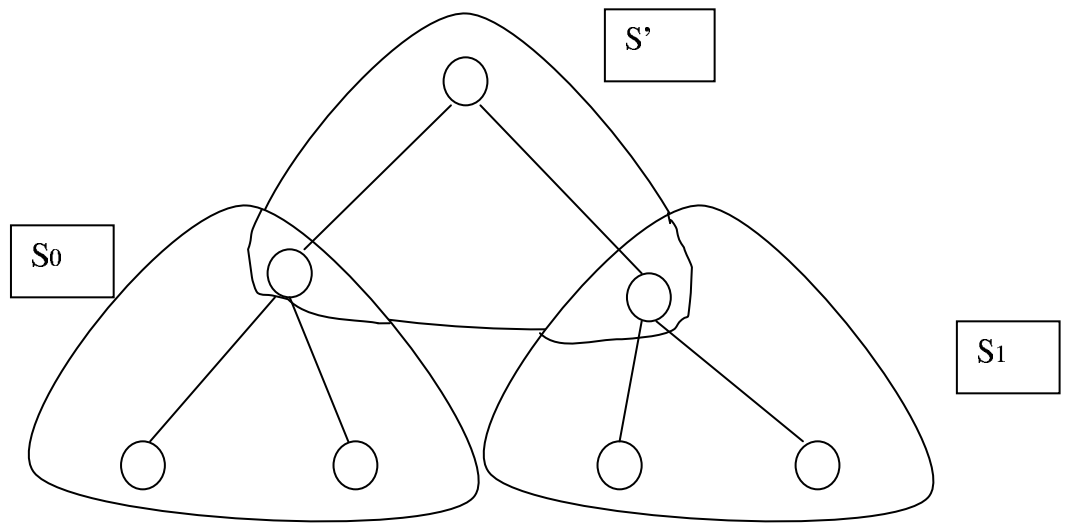


Figure 10

Here, we perform the operations by proceeding recursively.

Pred(S,x)

1. if $|x| = 1$, then,
2. if $x = 0$, then return \perp ,
3. else if $x = 1$ and $0 \notin S$ then return \perp
4. else if $x = 1$ and $1 \in S$ then return 0.
5. Let $x = x'x''$ where $|x'| = \lceil \frac{|x|}{2} \rceil$, $|x''| = \lfloor \frac{|x|}{2} \rfloor$.
6. If $x' \notin S'$, then,
7. $z' \leftarrow \text{Pred}(S', x')$
8. return $\max(z')$.
9. else if $x' \in S'$ and $x'' \leq \min(S_{x'})$, then,
10. $z' \leftarrow \text{Pred}(S', x')$
11. return $\max(z')$.
12. else if $x' \in S'$ and $x'' > \min(S_{x'})$, then,
13. $z'' \leftarrow \text{Pred}(S_{x'}, x'')$
14. return $x'z''$.
15. End.

Complexity Analysis: Each recursive call to $\text{Pred}(S,x)$ halves the length of $|x|$. Initially $|x| = \log m$. This gives $O(\log \log m)$ recursive calls. Thus, the total time is $O(q \log \log m)$, where q is the time needed to compute membership in S' . This may be $O(1)$, eg. if the dictionary data structure is a bit-vector. Thus the time complexity is $O(\log \log m)$.

Given a bit-vector dictionary, the data structure used for implementing the priority queue is called a stratified tree or a van Emde Boas tree [1].

Similarly, $\text{Find_Min}(S)$ also takes $O(\log \log m)$ or $O(\log h)$ time. However, the update operations Insert , Delete still take $O(\log m)$ time.

Goal: To improve the time complexity of $\text{Insert}(S,x)$ to $O(\log h)$.

Idea: We store $\min(S)$ and $\max(S)$ separately and represent only the remaining elements in the data structure.

Insert(S,x)

1. if $|s| = 0$ then,
2. $\min(S) \leftarrow x$,
3. $\max(S) \leftarrow x$,
4. return.
5. if $x > \max(S)$, then,
6. $x \leftrightarrow \max(S)$.
7. if $x < \min(S)$, then,
8. $x \leftrightarrow \min(S)$.
9. increment $|s|$
10. if $x \leq 2$ then, return
11. Let $x = x'x''$ where $|x'| = \lceil \frac{|x|}{2} \rceil$, $|x''| = \lfloor \frac{|x|}{2} \rfloor$
12. if $x' \in S$,
13. then, $Insert(S_{x'}, x'')$
14. else,
15. $\min(S_{x'}) \leftarrow x''$
16. $\max(S_{x'}) \leftarrow x''$
17. $Insert(S', x')$ i.e. insert x' into dictionary for S' .

This algorithm runs in $O(\log h) = O(\log \log m)$ time

Note: Here, $\min(S)$ and $\max(S)$ are stored separate from the data structure containing the remaining elements. Without this modification, $Insert(S, x)$ may require recursive computations of both $Insert(S', x')$ if $x' \notin S'$ and $Insert(S_{x'}, x'')$.

Therefore, the recurrence relation would be $T(h) = 2T(\frac{h}{2}) + 1$

$$\Rightarrow T(h) = O(h)$$

With modification, we have

$$T(h) = T(\frac{h}{2}) + 1$$

$$\Rightarrow T(h) = O(\log h).$$

To make this idea work for $Pred(S, x)$, we need to modify the definitions of S' and $S_{x'}$ as follows:

$$S' = \{x' | \exists x'', x'x'' \in S - \{\min(S), \max(S)\}\}$$

$$S_{x'} = \{x'' | x'x'' \in S - \{\min(S), \max S\}\}$$

Also, $\text{Pred}(S,x)$ is modified as follows:

Pred(S,x)

1. if $S = \phi$ or $x \leq \min(S)$
2. then return \perp
3. else if $x > \max(S)$
4. then return $\max(S)$, end.
5. Let $x = x'x''$ where $|x'| = \lceil \frac{|x|}{2} \rceil$, $|x''| = \lfloor \frac{|x|}{2} \rfloor$
6. if $x' \notin S'$
7. then, $z' \leftarrow \text{Pred}(S', x')$
8. if $z' = \perp$ then, return $\min(S)$
9. else return $z'.\max(S_{z'})$
10. else if $x' \in S'$ and $x'' \leq \min(S_{x'})$
11. then $z' \leftarrow \text{Pred}(S', x')$
12. if $z' = \perp$ then return $\min(s)$
13. else return $z'.\max(S_{z'})$
14. else if $x' \in S$ and $x'' > \min(S_{x'})$
15. then $z'' \leftarrow \text{Pred}(S_{x'}, x'')$
16. if $z'' = \perp$ then return $x'.\min(S_{x'})$
17. else return $x'.z''$

The complexity of the new $\text{Pred}(S,x)$ is still $O(\log h) = O(\log \log m)$.

References

- [1] Peter van Emde Boas, R. Kaas, E. Zijlstra, *Design and Implementation of an Efficient Priority Queue*, Mathematical Systems Theory 10: 99-127, 1977.