Advanced Data Structures
Lecture 5

Lecture by Dr. Shay Mozes
Typeset by Steven Karas

2016-12-08
Last edited 18:14:59 2016-12-28

1 Word RAM model

Words of size w, which can store a pointer or a value. $w > \log u$. An instruction on a word or pair of words takes $O(1)$. Following a pointer takes $O(1)$.

2 Predecessor/Successor Integer Problem

Given a set S of n integers from some universe $U = 1, ..., u-1$, where $n << u$

Operations

- member
- insert
- delete
- succ
- pred
- find-min/max

Various Implementations

f	Data structure	Time	Space
1962	Binary Trees	$O(\log n)$	$O(n)$
1975	van Emde Boas trees	$O(\log \log u)$	$O(u)$
????	x-fast trie	$O(\log \log u)$	$O(n \log u)$
1983	y-fast trie	$O(\log \log u)$	$O(n)$
1993	fusion trees	$O(\frac{\log n}{\log \log u})$[1]	$O(n)$

2.1 van Emde Boas Trees

The intuition is that we do a binary search on the representation, which gives us $O(\log \log n)$.

Basically, store 0 or 1 in a giant array. Then build a tree on top of it, where each node is 1 if any of the children are 1, and 0 is all of them are 0. Note that the depth of this tree is $O(\log u)$.

[1] Note that if $\log \log u = \omega(\sqrt{\log n})$, then we can sort integers in $O(n\sqrt{n})$

To reduce the size further, we store the tree as a bitfield. Note that for some word x, there are $\frac{1}{2}\log u$ high bits, and the same number of low bits. Note that there are $\sqrt{u}$ groups of the giant array with $\sqrt{u}$ elements in them. As such, we pack two lower trees into each word, and use higher layers to route to the lower trees.

Formally Recursive data structure with $\sqrt{u}+1$ sub-structures, each for a range $\sqrt{u}$. We refer to these as $S[0], ..., S[\sqrt{u}-1]$. We refer to the summary structure as $S.summary$. $S.summary[i] = 1$ iff $S[i]$ is not empty.

2.1.1 Membership

```
member(x):
  S[high(x)].member(low(x))
```

Complexity

$$T[\log u] = T\left[\frac{\log u}{2}\right] + \Theta(1) = \Theta(\log\log u)$$

2.1.2 Insertion

```
insert(x):
  S[high(x)].insert(low(x))
  S.summary.insert(high(x))
```

Complexity

$$T[\log u] = 2T\left[\frac{\log u}{2}\right] + \Theta(1) = \Theta(\log u)$$

A better way Store the first element explicitly (without making the recursive calls) in $S.min$.

```
insert(x):
  if S.min is nil:
    S.max := x
    S.min := x and return
  if x < S.min:
    swap x, S.min
  if x > S.max:
    S.max := x
  if S[high(x)].min is nil:
    S[high(x)].min := low(x)
    S[high(x)].max := low(x)
    S.summary.insert(high(x))
  else:
    S[high(x)].insert(low(x))
```

2.1.3 Successor

We also need to store the last element explicitly (without making the recursive calls) in $S.max$.

```
succ(x):
  if x < S.min:
    return S.min
  if low(x) < S[high(x)].max:
    return S[high(x)].succ(low(x))
  else
    succ_high = S.summary.succ(high(x))
    return S[succ_high].min | (succ_high * sqrt(u))
```

2.2 x-fast tries

Similar to van Emde Boas trees, but store the path as a sparse tree. We just need to find the closest node y that is 1 on the path from x to the root. If x is in the right subtree of y, pred(x) is the maximum in the left subtree. If x is in the left subtree, succ(x) is the minimum in the right subtree. Every node in this tree we will give a name which is the binary prefix path to the node for each element x in the set S.

We store this in a dynamic hash table, which requires $\Theta(n \cdot \log u)$.

Successor/Predecessor Find y by binary search on the path from x to the root (binary prefixes of x). This takes $O(\log \log u)$ time.

Insert Insert 1 for all the binary prefixes of x. This takes $O(\log u)$ amortized time.

Member Check the hash for x. $O(1)$

2.3 y-fast tries

Chunk into $n' = \frac{n}{\log u}$ groups of $\log u$ elements, store them in BSTs. Track roots of groups in x-fast trie. This gives us $\Theta(n' \cdot \log u) = \Theta(n)$ space, and the BSTs take $\frac{n}{\log u} \log u = \Theta(n)$ space. Note that the operations on the BSTs take $O(\log n)$. We need to ensure that the size of each of these BSTs is in $\left[\frac{1}{2} \log u,\ 2 \log u\right]$.

← There are $O(\log n)$ variants of all operations on BSTs, including split/merge

Member

Successor/Predecessor

Insertion

Indirection Compress successive elements of S into groups of $\Theta(\log u)$. Store the group as a BST, where all operations take $O(\log \log u)$. Store a representative of the group in a x-fast trie. The representative values are not required to even be in the groups.

Queries Given x - find the successor/predecessor y of x in the x-fast trie. Run the operation in the BST for y, or in the BST after y.

Insertion/Deletion Find the BST y for x, and run the operation on y. If y becomes too large, split it into y_1, y_2, and enter them into the x-fast trie. If y becomes too small, merge it with it's neighbor, potentially splitting the combined tree, and update the x-fast trie.

Note that insertions/deletions on the x-fast trie happen once every $\Theta(\log u)$ operations, and therefore take $\Theta(\log \log u)$ amortized time.

3 Self-adjusting Data Structures - Competitive Analysis

3.1 Competitive Analysis

Competitive Analysis compares our algorithm against one that has perfect knowledge. Is used in the Advanced Algorithms course to evaluate online algorithms.

Worst case $\Theta(n)$

Average case Access each element with uniform probability. $\Theta(n)$

Stochastic Access each element i with probability p_i such that $p_1 \geq p_2 \geq ... \geq p_n$. $\sum_{i=1}^{n} i \cdot p_i$

Static generic Sequence of queries $\sigma = x_1, ..., x_m$. Define f_i be the number of queries for element i. An optimal static solution is to order the elements by f_i descending.

Dynamic generic we are allowed to change the order of the list in runtime. The dynamic optimal solution is the most efficient solution given perfect knowledge.

Sleator Trajan Cost Model When we serach for the element x in the place i, we pay i. To move the element x to the front is free. Switching between neighbors costs us 1.

3.2 Linked Lists

A list of n elements $1, 2, ..., n$.

3.2.1 Access

Walk the list to find the element.

Possible strategy

Move to Front Move each element to the front when accessed.

Frequency Count Each time an element is accessed, reorder the list by frequency.

Transpose Promote on each access.

4 next week

splay trees