Thinking Points

How do we find (and remove) the smallest element in a ordered binary tree? What is the complexity?

Consider a modified linked list structure where node had an extra ‘next next’ pointer to the node two steps ahead. How would we search in this list? What would the complexity be?

Did we understand tree rotations?

Lecture 10
Advanced Data Structures Part 2
Heaps and Skip Lists

COMP26120

Giles Reger

December 2019

Last Time

Representing trees with array or linked structure

Lots of properties of trees

Breadth-first vs Depth-first

Binary search trees

Self-balancing AVL trees

Priority Queue ADT

Remember this...

A priority queue P supports the following fundamental operations:

`insert(k,e)`: Insert an element e with key k into P

`removeMin()`: Remove and return from P the element with the *smallest* key (may not be unique)

(error if P is empty)

Usually also support `size()` and `isEmpty()` operations.

How can we efficiently implement `removeMin()`?

(Binary) Heap

A (binary) heap is a complete (binary) tree with the heap-order property.

Definition (Heap-Order Property)

In a heap T, for every node v other than the root, the key stored at v is greater or equal to the key stored at v's parent.

Consequently, keys on any path from the root to an external node are in non-decreasing order.

Note that a complete tree always has a last node, defined as the right-most deepest node. Also, note that the height of a complete tree is always $\lceil \log(n+1) \rceil$, so if operations are $O(h)$ they are logarithmic in n.

Finally, because a heap is complete, it is efficient to use a vector representation.

Max-Heap vs Min-Heap

By default we assume we are defining a Min-Heap e.g. things got bigger as we go down the heap and we want to remove the minimum element.

The alternative is a Max-Heap. This straightforwardly reverses the order everywhere. Without loss of generality we just talk about Min-Heap but it is important to understand the difference.

Implementing `insert`

We need to preserve the heap-order and completeness properties.

To preserve completeness, we add the node as the last node. In a vector representation this will be the index immediately after the current last node (think about it). This is $O(1)$.

To preserve the heap-order property we then need to 'bubble' the new value up the heap until it is preserved. The general idea is that we recursively swap parent and child until the parent is smaller than the child again. This process is called bubble-up or sift-up.

This second step is $O(h)$, which we previously saw was $O(\log\ n)$.

Why do we only compare child and parent and not siblings?

Implementing `removeMin()`

Identifying the minimum element is now straightforward - it is always the root of the heap.

We now need to remove it whilst maintaining the heap-order and completeness properties.

We first replace the root with the last element. This preserves completeness.

But this will violate the heap-order property so we need to bubble-down or sift-down. To do this we compare a node to its children and if it is larger than both we terminate, otherwise we pick one, swap the parent and child and then repeat.

What is the complexity? Does it matter which child we pick?

Examples

Other Heap Thoughts

How can we can implement other container-like operations such as `find` and `remove`?

What about *melding* two heaps together?

What would change if we used a linked structure?

Other well-known heap variants include Binomial Heap and Fibonacci Heap. These are a bit more complex but give better guarantees for some operations.

Better Sorted Linked Lists

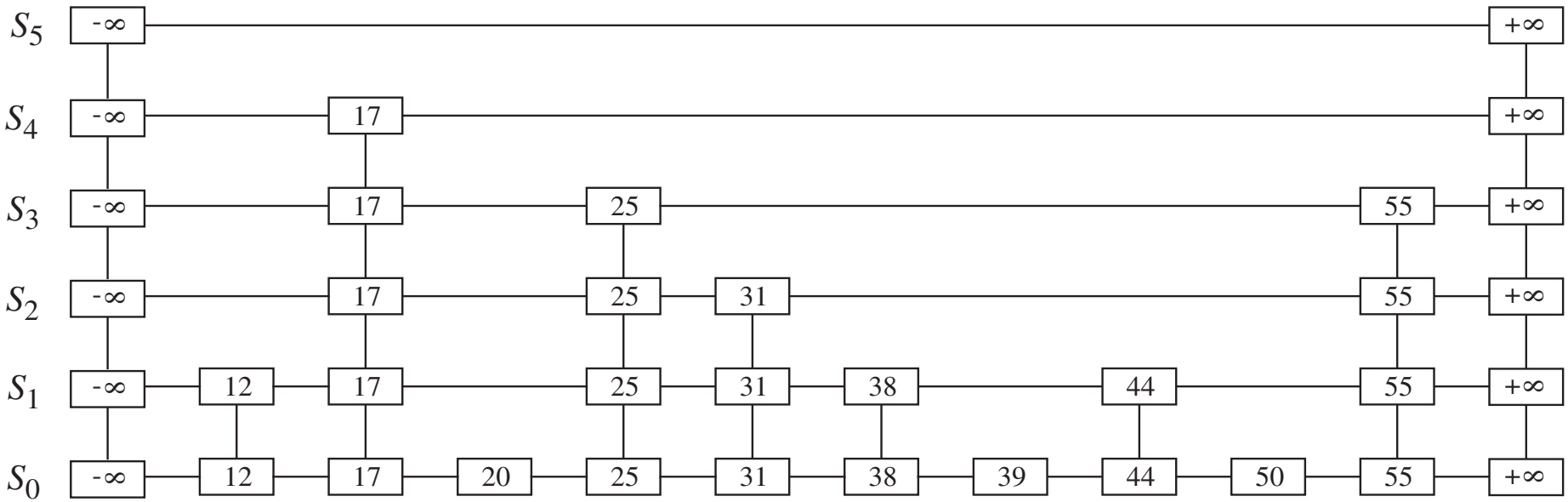
We cannot perform binary search in a sorted linked list but what if we had additional structure?

If we had a pointer into the middle of the list,
and then half way between each of those,
and then half way between each of those,
...

This is what a Skip List achieves

However, it is also a probabilistic data structure

Skip List



Nodes are arranged horizontally into levels and vertically into towers

Structure

A node contains a key-value pair and pointers to surrounding nodes.

Conceptually, each *node p* supports the operations

`after(p)` - return the node following p on the same level

`before(p)` - return the node preceding p on the same level

`above(p)` - return the node above p in the same tower

`below(p)` - return the node below p in the same tower

Nodes above/below have the same key-value pair. Keys are non-decreasing across the level (it is sorted).

In reality, we are more likely to represent towers as a separate data structure with a height and an array of next nodes.

Search for k

Effectively zig-zag down and right through the structure moving sideways when the next key is bigger than k and down otherwise

Start at 'top left' node p
While `below(p)` is not **null**
    p = `below(p)`
    While `key(after(p))` $\leq k$
        p = `after(p)`
Return p

Finds the bottom-level node with the largest key less-than-or-equal-to k

For `contains` can just check if `key(p)` $= k$

To ensure that the 'top left' node and `after`(p) always exist we always have two *sentinels* with keys $-\infty$ and ∞.

Insertion and Removal

To insert (k, v) we need to create a new node after `search`(k) at the bottom level and then decide how many additional levels to include the node in e.g. the height of the tower.

We do this randomly. Usually with probability $\frac{1}{2}$, e.g. flipping a coin, for each extra level.

We typically bound the maximum number of levels (e.g. to $3\lceil \log n \rceil$). What is the probability we go to a level higher than $\log n$?

Updating the actual nodes is synonymous to the linked list case.

For removal we simply remove the node $p =$ `search`(k) if `key`$(p) = k$ using the standard operations. Note that we never remove the sentinels.

Worst Case Complexity

As all operations are based on the search method (plus some local work) we just need to look at this.

The worst case is when every tower is of height $h - 1$. This collapses to a linked list where we also need to traverse down the last tower e.g. $O(n + h)$.

This is very unlikely.

Average Complexity

Firstly, the probability that the height h is greater than $3\log\ n$ is

$$\frac{n}{2^{3\log\ n}} = \frac{n}{n^3} = \frac{1}{n^2}$$

e.g. n chances to get $3\log\ n$ 'heads'.

We have high probability that h is $O(\log\ n)$.

The number of 'down' steps in the search method is h

What about scan-forward? We never return to a level. If we are using $\frac{1}{2}$ to decide whether to add a level then each key appears in 2 levels on average (expected number of times to flip a coin and get a sequence of 'heads').

Therefore, work per level is $O(1)$ and overall complexity is $O(h) = O(\log\ n)$

Space Complexity

The expected number of items at level i is $\frac{n}{2^i}$ meaning that the total number of items stored is

$$\sum_{i=0}^{h} \frac{n}{2^i} = n \sum_{i=0}^{h} \frac{1}{2^i} < 2n$$

Therefore, a skip list containing n items has a $O(n)$ space requirement.

Examples

Summary

Heap

- Heap-order property
- Complete binary tree (use vector representation)
- Bubble-up and bubble-down

Skip List

- Probabilistic data structure
- Levels of linked lists where each list contains roughly half the elements of the previous level
- Search by ‘zig-zagging’ down