UCT Algorithm Circle: Heaps

Gwylim Ashley

17 March 2011
Outline

1. Review of basics
2. Amortized analysis
3. Pairing heap
4. Binomial and Fibonacci heaps
5. Min-max heap
6. Other heaps
7. Summary
A heap is a tree-based data structure which has the *heap property* (the children of any node have keys that are \( \geq \) to the key of the parent)

- Heaps are used to implement *priority queues* (queues in which the element removed is always the minimum one)
- The heaps described are *min-heaps, max-heaps* have the larger elements at the top
A binary heap is a complete binary tree which satisfies the heap property.

We perform insertions and delete-mins using sift-up and sift-down operations.

Insertion and deletion of the minimum are $O(\log(n))$.

The binary heap can be efficiently stored in an array, since it is complete.
Decrease-key operation

- For certain algorithms (e.g. Dijkstra's path-finding algorithm), we may want to decrease the key of a node that is already in the heap.

- This operation can be implemented in various ways with different types of heaps, and a heap with an efficient implementation allows $O(|E| + |V| \log |V|)$ Dijkstra
Outline

1. Review of basics
2. Amortized analysis
3. Pairing heap
4. Binomial and Fibonacci heaps
5. Min-max heap
6. Other heaps
7. Summary
Amortized analysis

- When analyzing the complexity on an operation on a data structure, we usually consider an upper bound on the worst case $O(f(n))$ for some $f$.

- If the complexity of performing an operation $n$ times is $O(g(n))$, we define the *amortized complexity* of the operation to be $O(g(n)/n)$.

- This may not be the same as $O(f(n))$, as the worst case of the operation may be guaranteed not to occur repeatedly (e.g. if the operation ”improves” the data structure in some way).
Outline

1. Review of basics
2. Amortized analysis
3. Pairing heap
4. Binomial and Fibonacci heaps
5. Min-max heap
6. Other heaps
7. Summary
Pairing heap

- A pairing heap is a non-binary tree which satisfies the heap property.
- To merge two heaps, we can simply insert the one with the larger root as a child of the other (also providing a simple way to insert).
- Deletion of minimum is more complicated: we remove the root, then merge the children in pairs, and then merge the pairs recursively.
- This heap has better amortized insertion and merging than a binary heap, and is also conceptually simple.
- Decrease-key is $2^{\Theta(\sqrt{\log(\log(n))})}$ and $\Omega(\log(\log(n)))$. 
Outline

1. Review of basics
2. Amortized analysis
3. Pairing heap
4. Binomial and Fibonacci heaps
5. Min-max heap
6. Other heaps
7. Summary
Binomial tree

- A binomial tree of order 0 is a single node.
- A binomial tree of order \( k \neq 0 \) consists of a root with one child of each order \( k - 1, k - 2, \ldots, 0 \).
- Two binomial trees of order \( k - 1 \) can be combined to form one of order \( k \).
A binomial heap consists of a list of binomial trees with the heap property, sorted by order (with no two orders equal).

Merging heaps is done by merging trees of the same order from the heaps, traversing the heaps in a method similar to merge sort.
Fibonacci heap

- Similar to a binomial heap, but the trees do not have to be binomial trees (just satisfy the heap property)
- Merging (and hence insertion) are $O(1)$ since you can just add the linked lists together
- To delete the minimum, we need to merge the children of the minimum into the heap. At this point, we need to merge trees with the same number of children. This can be done in $O(log(n))$ amortized time
The main motivation for fibonacci heaps is $O(1)$ amortized decrease-key, which allows fast dijkstra.

In order to do this, we need to introduce *marking* of nodes.

A node can either be marked or unmarked.

When we decrease the key of a node, then if it becomes less than its parent, we cut it from its parent.

Then if its parent is marked, it is also cut (and becomes unmarked), otherwise it becomes marked.

This process is done recursively until we reach the root of the tree.
Example of fibonacci heap
Potential-based amortized analysis

- One way of doing amortized analysis makes use of a potential function on the data structure.
- The effective cost of an operation is given by the actual cost plus the change in potential, allowing us to take into account the effect on future operations.
- For the fibonacci heap, the potential is $t + 2m$, where $t$ is the number of trees, and $m$ is the number of marked nodes.
- In the decrease-key operation, if we create $k$ new trees, then the potential decreases by at least $k - 2$, which makes the decrease-key operation $O(1)$. 
Outline

1. Review of basics
2. Amortized analysis
3. Pairing heap
4. Binomial and Fibonacci heaps
5. Min-max heap
6. Other heaps
7. Summary
Double-ended priority queue?

- Suppose that we want a data structure allowing us to find and remove either the minimum or the maximum.
- One way of doing this is to have a min-heap and a max-heap which are linked together.
- However, we can also do this with one heap, a min-max heap, which is more space efficient.
Min-max heap

“[Min-max heaps are] min-max ordered: values stored at nodes on even (odd) levels are smaller (greater) than or equal to the values stored at their descendants (if any) where the root is at level zero.”
Sift-down for min-max heap

def sift_down(a):
    b ← minimum of children and grandchildren of a
    if b < a:
        swap a and b
        if b was a grandchild of a:
            if parent(b) < b:
                swap b and parent(b)
                sift_down(b)
def sift_up(a):
    if a is on min level:
        if a > parent(a):
            swap a and parent(a)
            sift_up_max(parent(a))
        else sift_up_min(a)
    else if a is on max level:
        if a < parent(a):
            swap a and parent(a)
            sift_up_min(parent(a))
        else sift_up_max(a)

def sift_up_min(a):
    if a < grandparent(a):
        swap a and grandparent(a)
        sift_up_min(grandparent(a))

sift_up_max is defined similarly
A soft heap is a probabilistic data structure which allows $O(1)$ delete-min, which is not possible with normal heaps.

In a soft heap with $n$ elements, at most $\epsilon n$ (where $0 < \epsilon < 1$) elements will be *corrupted* (have their key increased).

Thus an element may not be returned even if its key is actually the minimum.

The runtime of the *insert* operation is $O(\log(1/\epsilon))$, allowing us to choose the fraction of elements which is corrupted.

Soft heaps can be used to make a (non-random) $O(m\alpha(m, n))$ MST algorithm.
The Brodal queue is a parallel heap which has $O(1)$ for all operations.
For this structure, even arbitrarily long sorted lists can be inserted in $O(1)$.

Outline

1. Review of basics
2. Amortized analysis
3. Pairing heap
4. Binomial and Fibonacci heaps
5. Min-max heap
6. Other heaps
7. Summary

Gwylim Ashley  Heaps
<table>
<thead>
<tr>
<th>Type of heap</th>
<th>Find-min</th>
<th>Delete-min</th>
<th>Insert</th>
<th>Merge</th>
<th>Decrease-key</th>
</tr>
</thead>
<tbody>
<tr>
<td>Binary heap</td>
<td>$\Theta(1)$</td>
<td>$\Theta(\log(n))$</td>
<td>$\Theta(\log(n))$</td>
<td>$\Theta(n)$</td>
<td>$\Theta(\log(n))$</td>
</tr>
<tr>
<td>Pairing heap</td>
<td>$\Theta(1)$</td>
<td>$O(\log(n))$</td>
<td>$O(1)$</td>
<td>$O(1)$</td>
<td>$O(\log(n))$</td>
</tr>
<tr>
<td>Binomial heap</td>
<td>$\Theta(1)$</td>
<td>$O(\log(n))$</td>
<td>$O(1)$</td>
<td>$O(1)$</td>
<td>$O(1)$</td>
</tr>
<tr>
<td>Fibonacci heap</td>
<td>$O(1)$</td>
<td>$O(1)^{13}$</td>
<td>$O(1)^4$</td>
<td>$O(1)$</td>
<td>$O(1)^{13}$</td>
</tr>
<tr>
<td>Soft heap</td>
<td>$O(1)$</td>
<td>$O(1)^3$</td>
<td>$O(1)^4$</td>
<td>$O(1)$</td>
<td>$O(1)^4$</td>
</tr>
<tr>
<td>Brodal queue</td>
<td>$O(1)$</td>
<td>$O(1)^{13}$</td>
<td>$O(1)^{13}$</td>
<td>$O(1)^{13}$</td>
<td>$O(1)^{13}$</td>
</tr>
</tbody>
</table>

1 Amortized time
2 Parallel
3 Arbitrary delete
4 Takes a sorted sequence of elements