CSE 326 Autumn 2006
Section 3 Notes

1 Heap Summary

- In the table below, n represents the number of nodes in a heap, and we assume it is a min-heap everywhere, but the analysis is nearly identical for a max-heap.
- All heaps satisfy the constraint of *heap order*, that is, $key(parent) \leq key(child)$.
- The running time analysis is always worst-case, but amortized worst-case is averaged over *any* sequence of operations.
- The space requirements are those beyond what is required to store the key at each node, which is the same $O(n)$ for all heap types.
- Constant factors are given for comparison but are actually implementation-dependent.
- Logarithms are base 2 unless otherwise specified, but all logarithmic bases are asymptotically the same.
- Leftist and skew heaps are unbalanced in general, although specific cases can be balanced.
- INSERT is a special-case of MERGE with a one-node heap.
- DELETEMIN is a special-case of MERGE on the left and right subtrees after the root has been deleted.

Heap Type	**Representation**		**Balanced**	**Node Info**	**Space**
Binary Heap	Array		Yes	None	$O(1)$
d-Heap	Array		Yes	None	$O(1)$
Leftist Heap	Singly-linked nodes		No	npl	$O(2n)$
Skew Heap	Singly-linked nodes		No	None	$O(n)$
Binomial Queue	Forest of binary heaps		No	None	$O(n)$
Heap Type	**Insert**	**DeleteMin**	**Merge**	**BuildHeap**	**Analysis**
Binary Heap	$O(\log_2 n)$	$O(\log_2 n)$	$\Omega(n)$	$O(n)$	Worst-case
d-Heap	$O(\log_d n)$	$O(d \log_d n)$	$\Omega(n)$	$O(n)$	Worst-case
Leftist Heap	$O(2 \log_2 n)$	$O(2 \log_2 n)$	$O(2 \log_2 n)$	$O(n \log_2 n)$	Worst-case
Skew Heap	$O(\log_2 n)$	$O(\log_2 n)$	$O(\log_2 n)$	$O(n \log_2 n)$	Amortized
Binomial Queue	$O(\log_2 n)$	$O(\log_2 n)$	$O(\log_2 n)$	$O(n)$	Worst-case

Binary Heaps: A node with index i has a parent at $\lfloor i/2 \rfloor$ and children at $2i$ and $(2i+1)$, with the root at index 1. The merge time has a linear lower-bound $\Omega(n)$ simply because it requires $O(n)$ time to write the new merged array beginning with two separate arrays. INSERT and DELETEMIN use PERCOLATEUP and PERCOLATEDOWN, respectively, which in the worst-case requires traversal of the entire heap height, hence $O(\log_2 n)$.

The basic binary heap has an amortized BUILDHEAP time of $O(n)$, which consists of n inserts and no other operations in between. *Only in this case* can we assume that INSERT is $O(1)$.

d-Heaps: A node with index i has a parent at $\lfloor i/d \rfloor$ and children at di up to $(di + (d-1))$, with the root at index 1. However, this is not a compact representation in that it doesn't use all array positions. To make it compact, node i has a parent at $\lfloor (i-1)/d \rfloor$ and children at $(di+1)$, $(di+2)$, ..., $(di+d)$, with the root at index 0. In contrast to a binary tree with at most two children, a d-heap has at most d children, which lowers the INSERT time, but increases the DELETEMIN time because PERCOLATEDOWN must now compare keys with d children instead of 2. If d is constant, we can neglect this factor, but if we want to vary d to get better performance, we should be aware of this tradeoff.

Leftist Heaps: All operations are performed on the right-subtrees, which are kept short. This is done by keeping extra information at each node, the *null path length* (npl), or the length of the shortest path to a leaf. Leftist heaps improve on array-based heaps by allowing efficient merge but require a linked-list node representation with child pointers. Therefore, we have space $O(2n)$. The running times for MERGE, and therefore INSERT and DELETEMIN, have a factor of 2 because a check is required at each recursive level of the merge to maintain the *leftist property*: $npl(left) \geq npl(right)$.

Skew Heaps: Virtually identical to leftist heaps, but the npl check is omitted and the left and right subtrees are always swapped. This is based on empirical evidence that the subtrees are usually unbalanced after each recursive merge. Therefore, we no longer need to keep the extra information for each node. We are down to $O(n)$ for the child pointers and improve on the leftist heap's logarithmic running time by a constant factor.

Binomial Queues: An array (forest) of binary heaps are kept, with array position i corresponding to a binary heap of height i, called B_i. Some positions may be empty, but a binomial queue has at most one heap of any height. B_0 is a one-node heap. The *binomial queue property* is that B_{k+1} is formed by making $B_0, \ldots, B_k$ subtrees of a new root node. Merging binomial queues is analogous to adding binary numbers, and there are $O(\log_2 n)$ binary heaps in each forest. Binomial queues improve on leftist and skew heaps by having an amortized worse-case BUILDHEAP time (insertions with no deletions) of $O(1)$ while still keep $O(\log_2 n)$-time merge operations.

2 Sum identities

Because I ran out of time, here are the full proofs of the sums I intended to do plus a few miscellaneous goodies. But first, recall the following useful identities:

$$\sum_{i=0}^{n} i = \frac{n(n+1)}{2} \tag{1}$$

$$\sum_{i=m}^{n} f(i) = \left[\sum_{i=0}^{n} f(i)\right] + \left[\sum_{i=m}^{n} f(i)\right] \tag{2}$$

$$\tag{3}$$

The following nested sums are useful in solving the running times of nested loops.

3 Sum of squares

First, let's write out the first couple of integer squares and draw them pictorially in a suggestive way.

1	4	9	16	...
□	□□	□□□	□□□□	...
	□□	□□□	□□□□	...
		□□□	□□□□	...
			□□□□	...

What I mean by "suggestive" is that if we sum the first row of blocks, this is just the sum of the first n integers starting from 1 (and Gauss already told us how to solve this with the identity above!). If we sum the second row of blocks, this is just the same sum but starting from 2 (and the other identity above tells us how to start from a number other than 1). Likewise, the sum of the third row of blocks is the same sum starting from 3, and so on. We can then write the sum of squares as a double sum of non-square integers, and then simplify using identities from above.

$$
\begin{aligned}
f(n) &= \sum_{i=1}^{n} i^2 && (4)\\
&= \sum_{i=1}^{n}\sum_{j=i}^{n} j && (5)\\
&= \sum_{i=1}^{n}\left[\left(\sum_{j=1}^{n}\right) - \left(\sum_{j=1}^{i-1}\right)\right] && (6)\\
&= \sum_{i=1}^{n}\left[\frac{n(n+1)}{2} - \frac{(i-1)i}{2}\right] && (7)\\
&= \frac{n^2(n+1)}{2} - \frac{1}{2}\sum_{i=1}^{n} i^2 + \frac{1}{2}\sum_{i=1}^{n} i && (8)\\
&= \frac{n^2(n+1)}{2} - \frac{1}{2}f(n) + \frac{n(n+1)}{2} && (9)\\
\frac{3}{2}f(n) &= \frac{n^2(n+1)}{2} + \frac{n(n+1)}{4} && (10)\\
&= \frac{2n^2(n+1) + n(n+1)}{4} && (11)\\
&= \frac{n(2n+1)(n+1)}{4} && (12)\\
f(n) &= \frac{n(2n+1)(n+1)}{6} && (13)
\end{aligned}
$$

4 Sum of $\frac{i}{2^i}$

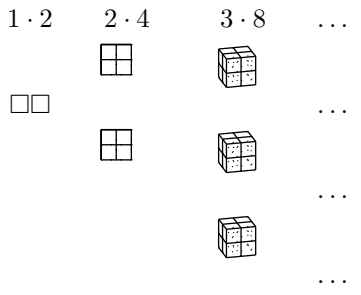
I incorrectly told some of you that this sum did not converge to a constant in section, when in fact it does. You can use the example shown in Chapter 1 of the text to compute this by expanding the sum, multiplying both sides by a constant, and then subtracting to get a known sum.

5 Running time of basic BuildHeap

As an alternative to the method shown in Chapter 6, pages 211-214, where h is the height of the heap and we sum over each level i.

$$
\begin{aligned}
S &= \sum_{i=0}^{h} 2^i(h-i) && (14)\\
&= h\sum_{i=0}^{n} 2^i - \sum_{i=0}^{h} i2^i && (15)\\
&= h\frac{2^{h+1}-1}{2-1} - \sum_{i=0}^{h} i2^i && (16)
\end{aligned}
$$

Again, we draw a suggestive picture to help us transform $i2^i$ into a double sum of something we already know how to solve. This is the same meaning of "suggestive" as the previous problem.



$$
\begin{aligned}
T(n) &= h\frac{2^{h+1}-1}{2-1} - \sum_{i=0}^{h}\sum_{j=i}^{h} 2^j && (17)\\
&= h2^{h+1} - h - \sum_{i=0}^{h}\left(\left[\sum_{j=0}^{h} 2^j\right] - \left[\sum_{j=0}^{i-1} 2^j\right]\right) && (18)\\
&= h2^{h+1} - h - \sum_{i=0}^{h}\left(\frac{2^{h+1}-1}{2-1} - \frac{2^i-1}{2-1}\right) && (19)\\
&= h2^{h+1} - h - \left(\sum_{i=0}^{h} 2^{h+1} - 2^i\right) && (20)\\
&= h2^{h+1} - h - h2^{h+1} + \frac{2^{h+1}-1}{2-1} && (21)\\
&= h2^{h+1} - h - h2^{h+1} + 2^{h+1} - 1 && (22)\\
&= 2^{h+1} - h - 1 && (23)
\end{aligned}
$$

This is off by one from what the book got, so I missed something somewhere. Meh, close enough. If you were off by one on a test, you'd probably still get most of the credit. See if you can find my mistake!

Anyway, as I was saying:

$$
\begin{aligned}
h = \log_2 n \quad &\Rightarrow \quad T(n) = 2^{\log_2 n + 1} - \log_2 n - 1 && (24)\\
&\Rightarrow \quad T(n) = O(n) && (25)
\end{aligned}
$$

6 Size of a binomial queue

Why do we assume that a binomial queue has $O(\log n)$ heaps? By analogy with binary numbers, the number n has $\log_2 n$ bits. By a more rigorous sum, we bound the number of nodes that can be contained in a binomial queue of a certain size, m.

$$
\begin{aligned}
\sum_{i=0}^{m} 2^i &\leq n && (26)\\
\frac{2^{m+1}-1}{2-1} &\leq n && (27)\\
2^{m+1} &\leq n+1 && (28)\\
m &\leq \log_2(n+1) - 1 && (29)\\
& && (30)
\end{aligned}
$$

Therefore, the number of trees in the forest, m, is upper-bounded by $\log_2 n$.