Design and Analysis of Algorithms II: Heaps
Lecture 2

Nick Ferrara

September 14th, 2023

Contents

1 Dynamic Arrays with Load Factor

1.1 Operations

The primary operations considered are:

- **Insert**: Add an element to the array.
- **Delete**: Remove an element from the array.

Let S be the size of the array, and n be the number of elements in the array.

1.2 Load Factor

The load factor, denoted as α, is defined as the ratio $\frac{n}{s}$.

1.2.1 Desirable Load Factor

When deciding when to grow and shrink the array, we aim to keep the load factor within some constant limits to ensure reasonable performance for insert and delete operations. It is reasonable to avoid choosing a constant limit that is too close to $\frac{1}{2}$, since we may encounter situations where the array grows and shrinks frequently between consecutive insertions and deletions, which would inflate unnecessary costs.

1.2.2 Acceptable Range

A reasonable range for α is $\frac{1}{4} \leq \alpha \leq 1$ where we shrink the array if $\alpha = \frac{1}{4}$ and double the array when $\alpha = 1$.

1.3 Amortized Analysis

Insertions have a $\Theta(1)$ amortized cost in this case.

$$\phi = \begin{cases} 2n - S, & \text{if } \alpha \geq \frac{1}{2}, \\ S/2 - n, & \text{if } \alpha < \frac{1}{2}. \end{cases} \tag{1}$$

1.3.1 Further Reading

More detailed analysis covering the cases for doubling and shrinking is available in the reference book.

2 Priority Queues (Min-PQ)

2.1 Definition and Properties

A priority queue (Min-PQ) is a data structure that allows efficient handling of elements based on their priorities. The element with the lowest priority can be accessed or removed quickly.

2.2 Operations

The main operations associated with a min-priority queue are:

- **Insert**(x, data): Insert an element with key x and its associated data into the priority queue.
- **ExtractMin**(): Remove and return the element with the smallest priority.
- **FindMin**(): Identify the element with the smallest priority without changing the structure.
- **DecreaseKey**(item, newkey): Decrease the key of an existing item, then restore the heap by bubbling up.
- **Delete**(item): Delete an item from the heap.

2.2.1 Time Complexity

For a min-priority queue implemented as a binary heap, all operations have a time complexity of $\Theta(\log n)$ except for FindMin(), which operates in $\Theta(1)$ time, where n is the number of elements in the priority queue.

2.3 Examples and Applications

Some applications of min-priority queues are:

- Prim's Minimum Spanning Tree (MST) algorithm
- Dijkstra's shortest path algorithm

2.4 Meldable/Mergable Priority Queues

These types of priority queues allow merging operations. For example, merging binary heaps has a time complexity of $\Theta(n)$.

2.5 Can We Do Better?

Yes, performance can be improved using data structures like Binomial Heaps where all operations have a $\Theta(\log n)$ cost except for FindMin(), which operates in $\Theta(1)$ time. We will discuss Binomial Heaps in a later section.

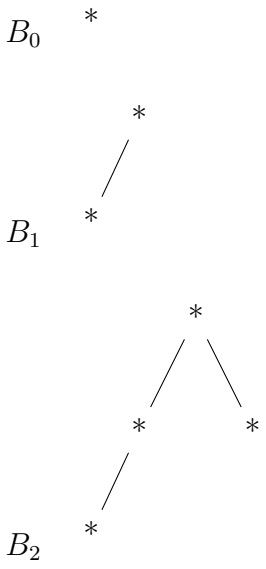
3 Binomial Trees

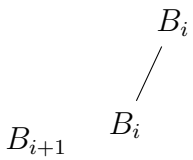
3.1 Ordered Trees

In binomial trees, children are ordered. We denote them as $B_0, B_1, B_2, \ldots$.

3.2 Understanding The Recursive Structure

Binomial trees are built recursively. We start with B_0, which consists of a single node, and we build B_{i+1} by connecting a copy of B_i as the leftmost child of another B_i.





3.3 Properties of B_k Trees

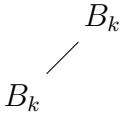
- Size $= 2^k$
- Height $= k$
- Number of items on level i is $\binom{k}{i}$
- Root degree is k with degrees of its children being $k-1, k-2, \ldots, 0$ in order from left to right.

Example for B_0: The B_0 tree consists of a single node.

$$\text{Size}_{B_0} = 1 = 2^0$$

$$\text{degree}_{B_0} = 0$$

Example for B_{k+1}: The B_{k+1} tree can be formed by attaching a B_k sub tree as the left child of another B_k sub tree.



$$\text{Size}_{B_{k+1}} = 2 \times \text{Size}_{B_k} = 2 \times 2^k = 2^{k+1}$$

$$\text{degree}_{B_{k+1}} = 1 + \text{degree}_{B_k} = k + 1$$

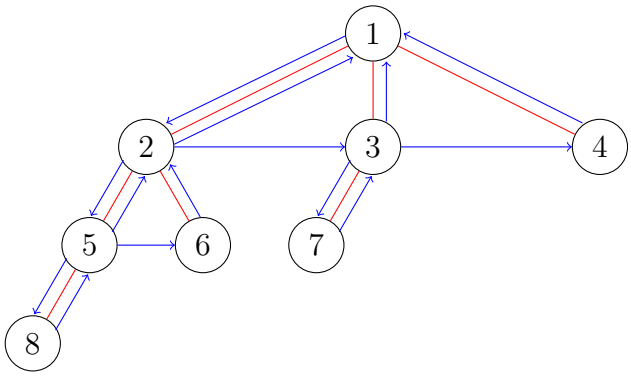
$$\text{height}_{B_{k+1}} = 1 + \text{height}_{B_k} = k + 1$$

$$\#\text{ nodes at level } j \text{ in } B_{k+1} =$$

$$\#\text{ nodes at level } j-1 \text{ in } B_k + \#\text{ nodes at level } j \text{ in } B_k = \binom{k}{j-1} + \binom{k}{j} = \binom{k+1}{j}$$

3.4 Example Tree Structure

The binomial tree is structured in the (Left-child, Right-sibling) form as seen below.



The parent-child and sibling relationships are as follows:

- 1: left child is 2
- 2: parent is 1, right sibling is 3, left child is 5
- 3: parent is 1, right sibling is 4, left child is 7
- 4: parent is 1
- 5: parent is 2, right sibling is 6, left child is 8
- 6: parent is 2
- 7: parent is 3
- 8: parent is 5

4 Heap-Ordered Trees (Min-Heap)

4.1 Introduction to Min-Heaps

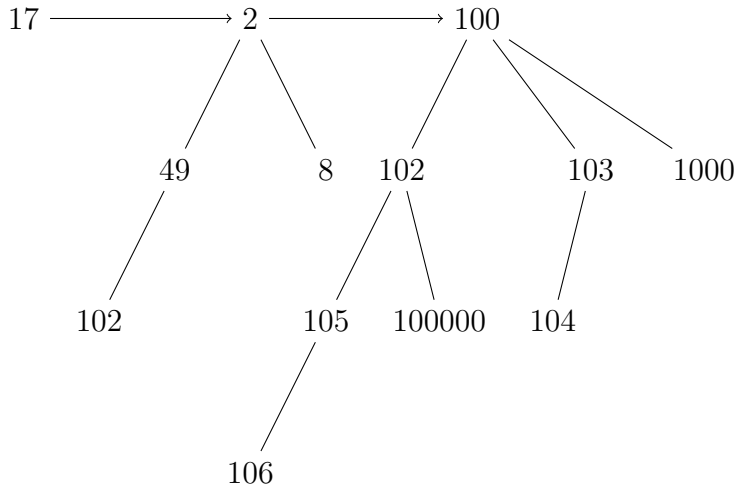
A min-heap is a tree-based data structure in which each parent node has a key that is less than or equal to the keys of its children. Formally, for each node N with parent node P, the key of P is less than or equal to the key of N.

4.2 Binomial Heaps

4.2.1 Introduction

A binomial heap is a specific type of min-heap that is made up of a collection of heap-ordered binomial trees. Each binomial heap is represented as a linked list of the roots of binomial trees, and these trees are ordered by increasing degree. Additionally, no two trees in the heap have the same degree.

Example of a Binomial Heap To provide a concrete example, consider the following structure, where arrows represent the linked list of root nodes, and the tree structures represent heap-ordered binomial trees:



In our example, the root nodes are 17, 2, and 100, and these form a linked list. The tree rooted at 2 has children 49 and 8, where 49 further has a left child 102. The tree rooted at 100 is more complex and contains nodes 102, 103, and 1000 as its immediate children, each of which have their own sub-trees.

4.2.2 Understanding Binomial Heap Structure

Suppose the binomial heap contains $n = 13$ elements. We can represent n as a binary number to identify the degrees of the trees present in the binomial heap. Specifically, $n = 13 = 1101_2$ in binary. Here, each bit represents whether a tree of a certain degree is present (1) or not (0) in the heap. In this way, each bit can be thought of as corresponding to a tree of a degree that is present in the heap. This representation is useful because each tree in a binomial heap has a size that is a power of 2, which aligns with the binary representation of n.

Another point to note is that there are at most $\log_2(n)$ trees in the binomial heap. This makes sense since each bit in the binary representation of n represents a unique tree in the heap, and there can be at most $\log_2(n)$ bits for a number n.

4.2.3 Operations on Binomial Heaps

- **Link(T_1, T_2):** Merges two binomial trees of the same degree into a new binomial tree of degree one higher. Specifically, it links two trees where degree(T_1) = degree(T_2). The tree rooted at the node with the smaller key becomes the new root, and the other tree becomes its leftmost child. This operation has a time complexity of $\Theta(1)$.

- **List-Merge(H_1, H_2):** merges two lists of binomial trees, H_1 and H_2, into a single list in ascending degree order. Importantly, it is not used as a standalone operation; rather, it is a "dummy" operation serving as a subroutine for the Union operation. The operation cost is $\Theta(\log n_1 + \log n_2) = \Theta(\log n)$ where n_1 and n_2 are the number of elements in each heap.

- **Union(H_1, H_2):** Uses List-Merge as a subroutine to initially produce an invalid binomial heap. Subsequently, the operation goes through the list, linking any trees with the same degree to produce a valid binomial heap. The total cost includes the $\Theta(\log n)$ cost from List-Merge and $\Theta(\log n)$ from linking, making the total operation cost $\Theta(\log n)$.

- **Insert(Tree, H_0):** Converts a single binomial tree into a one-element binomial heap, denoted H_{Tree}, and then merges it with H_0 using the Union operation. The total cost is $\Theta(\log n)$.

- **ExtractMin():** Identifies and removes the tree with the smallest root. The children of this tree are then treated as a new descending order binomial heap. Using Union, this new heap is merged back with the original one. The total cost of this operation is $\Theta(\log n)$.

- **DecreaseKey(item, key):** Reduces the item's key, and then the item is "bubbled up" by swapping with its parent until it either becomes the root or its parent has a smaller key. The operation's cost is $\Theta(\log n)$ as the item can climb at most the height of the tree, which is at most $\log n$.

- **Delete(item):** Uses DecreaseKey to reduce the key to a value smaller than any other in the heap (e.g., $-\infty$) and then employs ExtractMin() to remove it. Both operations cost $\Theta(\log n)$, making the total cost $\Theta(\log n)$.

4.3 Fibonacci Heaps

4.3.1 Introduction

In Fibonacci heaps, all operations have an amortized cost of $\Theta(1)$ except for ExtractMin() and DecreaseKey(), which have an amortized cost of $\Theta(\log n)$.

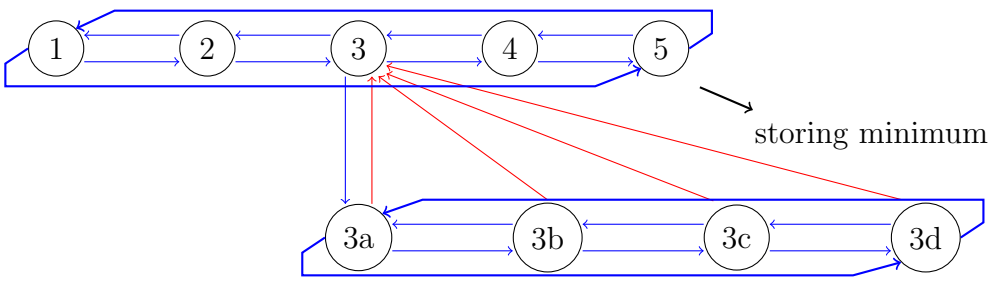
Let $D(n)$ be the maximum degree of any node in any Fibonacci heap of size $n \leq n$.

Nodes in Fibonacci heaps can be marked with a bit, either on or off.

Let $\phi(H) = t(H) + 2 \times m(H)$ be the potential function we use for Fibonacci heaps, where t is the size of the tree root list and m is the number of marked nodes in the heap.

4.3.2 Understanding Fibonacci Heap Structure

Fibonacci heaps have a heap-ordered root list with its children unordered. The minimum of the heap is stored such that it can be accessed in constant time. The root list, along with all levels of the trees, are doubly linked lists that are cyclic. Below, we illustrate an example of a Fibonacci heap to understand this description.



4.3.3 Operations on Fibonacci Heaps

- **Insert(x):** Add a small tree {x} to the root list, updating the minimum as necessary. This has an amortized cost of $\Theta(1)$.

- **Merge(H_1, H_2):** Simply concatenate the root lists of H_1 and H_2, updating the minimum as necessary. This has an amortized cost of $\Theta(1)$.
- **ExtractMin():** The children heap of the minimum root is merged with the original root list, and then an operation called *consolidate* is executed, which will be discussed in the next lecture. This has an amortized cost of $\Theta(\log n)$.

The rest of the Fibonacci heaps will be covered in the next lecture.