Kattis

Heap

Arnar Bjarni Arnarson

September 21, 2025

School of Computer Science
Reykjavík University

Heaps

- As an example of a data structure in the standard library but that sometimes requires a more powerful version, let us consider heaps.

Heaps

- As an example of a data structure in the standard library but that sometimes requires a more powerful version, let us consider heaps.
- Heaps are implemented in most standard libraries in the forms of priority queues.

Heaps

- As an example of a data structure in the standard library but that sometimes requires a more powerful version, let us consider heaps.
- Heaps are implemented in most standard libraries in the forms of priority queues.
- A heap is nothing but a binary tree satisfying *the heap condition*.

Heaps

- As an example of a data structure in the standard library but that sometimes requires a more powerful version, let us consider heaps.
- Heaps are implemented in most standard libraries in the forms of priority queues.
- A heap is nothing but a binary tree satisfying *the heap condition*.
- The heap condition (for a min heap) says that the value of any given node is not greater than that of its children.

Heaps

- Since arrays are linear, we want to smush this binary tree into an array for the implementation.

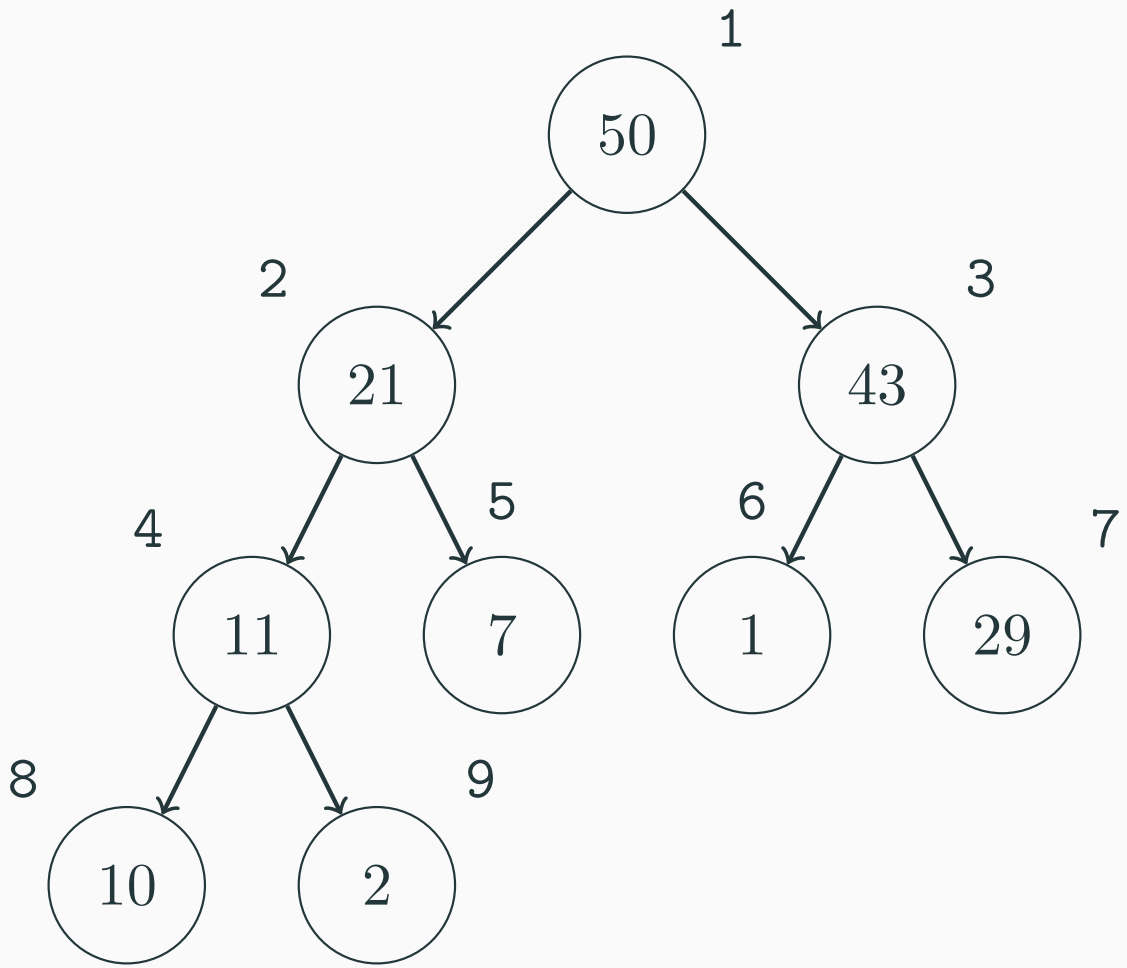
Heaps

- Since arrays are linear, we want to smush this binary tree into an array for the implementation.
- We can do this by putting the root at index 1.
- Then the children of item at index i are simply at $2i$ and $2i + 1$.
- The parent of any item $i > 1$ is then $\left\lfloor \frac{i}{2} \right\rfloor$.

Heaps

- Since arrays are linear, we want to smush this binary tree into an array for the implementation.
- We can do this by putting the root at index 1.
- Then the children of item at index i are simply at $2i$ and $2i + 1$.
- The parent of any item $i > 1$ is then $\left\lfloor \frac{i}{2} \right\rfloor$.
- We could do this using raw arrays (then index 0 can be used to store its size), but the examples will be given in C++ using vectors.

Heaps



`ARRAY: [SIZE, 50, 21, 43, 11, 7, 1, 29, 10, 2]`

Heaps

- Items can be inserted by pushing them to the back and fixing the heap condition upwards from them.

Heaps

- Items can be inserted by pushing them to the back and fixing the heap condition upwards from them.
- Items can be deleted by replacing the smallest value with a leaf and then fixing the heap condition downwards.

Heaps

- Items can be inserted by pushing them to the back and fixing the heap condition upwards from them.
- Items can be deleted by replacing the smallest value with a leaf and then fixing the heap condition downwards.
- Let us see how this would look in C++.

C++ implementation (min-heap)

```
template<typename T> struct Heap {
    vector<T> h; Heap() : h(1) { }
    constexpr size_t size() { return h.size() - 1; }
    constexpr T peek() { return h[1]; }
    void swim(size_t i) {
        while(i != 1 && h[i] < h[i / 2]) {
            swap(h[i], h[i / 2]);
            i /= 2; } }
    void sink(size_t i) {
        while(true) {
            size_t mn = i;
            if(2 * i + 1 < h.size() && h[mn] > h[2 * i + 1]) mn = 2 * i + 1;
            if(2 * i  < h.size() && h[mn] > h[2 * i]) mn = 2 * i;
            if(mn != i) swap(h[i], h[mn]), i = mn;
            else break; } }
    void pop() {
        h[1] = h.back();
        h.pop_back(); sink(1); }
    void push(T x) {
        h.push_back(x);
        swim(h.size() - 1); } };
```

Heaps

- We note that peek and size run in $\mathcal{O}(1)$ while all other operations run in $\mathcal{O}(log(n))$.

Heaps

- We note that peek and size run in $\mathcal{O}(1)$ while all other operations run in $\mathcal{O}(log(n))$.
- This implementation isn't any better than the standard library one in C++.

Heaps

- We note that peek and size run in $\mathcal{O}(1)$ while all other operations run in $\mathcal{O}(log(n))$.
- This implementation isn't any better than the standard library one in C++.
- We provide it for demonstration of representing binary trees with an array.