Another set data structure

Idea: store data in a collection of arrays
- array i has size 2^i
- an array is either full or empty (never partially full)
- each array is stored in sorted order
- no relationship between arrays

Binary array set
https://www.nayuki.io/page/binary-array-set

Binary heap

A binary tree where the value of a parent is greater than or equal to the value of its children

Additional restriction: all levels of the tree are complete except the last

Max heap vs. min heap
Binary heap - operations

- **Max** - return the largest element in the set
- **ExtractMax** - Return and remove the largest element in the set
- **Insert(val)** - insert val into the set
- **IncreaseElement(x, val)** - increase the value of element x to val
- **BuildHeap(A)** - build a heap from an array of elements

Binary heap representations

1. Assume left and right children are heaps, turn current node into a valid heap

```
Heapify(A, i)
1. i ← LEFT(i)
2. r ← RIGHT(i)
3. largest ← i
5.     largest ← r
7.     largest ← r
8. IF largest ≠ i
9.     swap A[i] and A[largest]
10.    Heapify(A, largest)
```

2. Assume left and right children are heaps, turn current set into a valid heap

```
Heapify(A)
1. i ← LEFT(i)
2. r ← RIGHT(i)
3. largest ← i
5.     largest ← r
7.     largest ← r
8. IF largest ≠ i
9.     swap A[i] and A[largest]
10.    Heapify(A, largest)
```
**Heapify**

Assume left and right children are heaps, turn current set into a valid heap.

```plaintext
Heapify(A, i)
1   i ← Left(i)
2   r ← Right(i)
3   largest ← i
5       largest ← L
7       largest ← R
8   if largest ≠ i
9       swap A[i] and A[largest]
10  Heapify(A, largest)
```

If a child is larger, swap and recurse.
Running time of Heapify

O(height of the tree)

What is the height of the tree?
- Complete binary tree, except for the last level

2^h ≤ n
h ≤ log₂ n
O(log n)

Binary heap - operations

Max - return the largest element in the set

Max
What is the largest element in the set?
Return A[1]

ExtractMax - Return and remove the largest element in the set

ExtractMax
Return and remove the largest element in the set

Insert(val) – insert val into the set

Insert(val)

IncreaseElement(x, val) – increase the value of element x to val

IncreaseElement(x, val)

BuildHeap(A) – build a heap from an array of elements

BuildHeap(A)
ExtractMax
Return and remove the largest element in the set

Heapify

ExtractMax
Return and remove the largest element in the set

IncreaseElement (aka swim up)
Increase the value of element $x$ to $val$

Heapify
IncreaseElement
Increase the value of element x to val

IncreaseElement
Increase the value of element x to val

IncreaseElement
Increase the value of element x to val

IncreaseElement
Increase the value of element x to val
**IncreaseElement**

Increase the value of element $x$ to $val$

- Runtime: $O(\text{height of tree}) = O(\log n)$

**Insert**

Insert $val$ into the set

- Propagate value up
Building a heap

Can we build a heap using the functions we have so far?
- Max
- ExtractMax
- Insert(val)
- IncreaseElement(x, val)

Building a heap

For each element x in array:
insert(x)

Running time of BuildHeap1

n calls to Insert – O(n log n)

Can we do better?
Building a heap: take 2

**Build-Heap(A)**

1. **heap-size[A] = (length[A])**
2. For **i = [(length[A])/2]↓**
3. **Heapify(A, i)**

Start with \( n/2 \) "one-node" heaps

call Heapify on element \( n/2-1, n/2-2, n/2-3 \ldots \)

all children have smaller indices

building from the bottom up, makes sure that all the children are heaps
Running time of BuildHeap2

$n/2$ calls to Heapify $-$ $O(n \log n)$

Can we get a tighter bound?
Running time of BuildHeap2

How many nodes are at level $d$? $2^d$

Running time of BuildHeap2

$T(n) = \sum_{i=1}^{\log n} 2^i$ (cost)

Nodes at height $h$

- $h < \lceil \log(n/2) \rceil$ nodes
- $h=0 < \lceil \log(n/2) \rceil$ nodes
- $h=1 < \lceil \log(n/4) \rceil$ nodes
- $h=2 < \lceil \log(n/8) \rceil$ nodes

Running time of BuildHeap2

$T(n) = \sum_{h=0}^{\log n} \lceil \log(n/2^h) \rceil$

- $= \lceil \log(n/2^0) \rceil$
- $= \lceil \log(n/2^1) \rceil$
- $= \lceil \log(n/2^2) \rceil$
- $= \Theta(n)$
Binary heaps

- Procedure
  - Binary heap (worst-case)
  - BUILD-HEAP: $\Theta(n)$
  - INSERT: $O(\log n)$
  - MAXIMUM: $O(1)$
  - EXTRACT-MAX: $O(\log n)$
  - UNION: $O(1)$
  - INCREASE-ELEMENT: $O(\log n)$
  - DELETE: $O(\log n)$
  (adapted from Figure 19.1, pg. 456 [3])

Mergeable heaps

- Procedure
  - Binary heap (worst-case)
  - BUILD-HEAP: $O(n)$
  - INSERT: $O(\log n)$
  - MAXIMUM: $O(1)$
  - EXTRACT-MAX: $O(\log n)$
  - UNION: $O(\log n)$
  - INCREASE-ELEMENT: $O(\log n)$
  - DELETE: $O(\log n)$
  (adapted from Figure 19.1, pg. 456 [3])

- Mergeable heaps support the union operation
- Allows us to combine two heaps to get a single heap
- Union runtime for binary heaps?

Union for binary heaps

- Procedure
  - Binary heap (worst-case)
  - BUILD-HEAP: $O(n)$
  - INSERT: $O(\log n)$
  - MAXIMUM: $O(1)$
  - EXTRACT-MAX: $O(\log n)$
  - UNION: $O(\log n)$
  - INCREASE-ELEMENT: $O(\log n)$
  - DELETE: $O(\log n)$
  (adapted from Figure 19.1, pg. 456 [3])

- Concatenate the arrays and then call Build-Heap

Linked-list heap

- Store the elements in an unordered doubly linked list
- Insert:
- Max:
- Extract-Max:
- Increase:
- Union:
Linked-list heap

Store the elements in an unordered doubly linked list

- Insert: add to the end/beginning
- Max: search through the linked list
- Extract-Max: search and delete
- Increase: increase value
- Union: concatenate linked lists

Faster Union, Increase, Insert and Delete… but slower Max operations
Binomial Tree

Height?

\[ H(B_0) = 1 \]
\[ H(B_k) = 1 + H(B_{k-1}) = k \]

Binomial Tree

Degree of root node?

\[ k, \text{ each time we add another binomial tree} \]

Binomial Tree

What are the children of the root?

\[ k \text{ binomial trees: } B_{k-1}, B_{k-2}, \ldots, B_0 \]

Binomial Tree

Why is it called a binomial tree?

- depth 0
- depth 1
- depth 2
- depth 3
- depth 4
Binomial Tree

$B_k$ has $2^i$ nodes at depth $i$.

Another set data structure: recap

Idea: store data in a collection of arrays
- array $i$ has size $2^i$
- an array is either full or empty (never partially full)
- each array is stored in sorted order
- no relationship between arrays

Another set data structure: recap

Which arrays are full and empty are based on the number of elements
- specifically, binary representation of the number of elements
- 4 items = 100 = $A_2$-full, $A_1$-empty, $A_0$-empty
- 11 items = 1011 = $A_3$-full, $A_2$-empty, $A_1$-full, $A_0$-full

Lookup: binary search through each array
- Worst case runtime?

Binomial Heap

Binomial heap was introduced by Vuillemin in 1978.

Sequence of binomial trees that satisfy binomial heap property:
- each tree is min-heap ordered
- top level: full or empty binomial tree of order $k$
- which are empty or full is based on the number of elements
Binomial Heap

- $A_h$: [18]
- $A_{h-1}$: [3, 7]
- $A_{h-2}$: empty
- $A_{h-3}$: empty
- $A_{h-4}$: [6, 8, 29, 10, 44, 30, 23, 22, 48, 31, 17, 45, 32, 24, 55]

$N = 19$

- $\#$ trees = 3
- height = 4
- binary = 10011

How many heaps?

- $O(\log n)$ – binary number representation

Where is the max/min?

- Must be one of the roots of the heaps

Runtime of max/min?

- $O(\log n)$
Binomial Heap: Properties

Height?

\[ \log_2 n \]
- largest tree = \( B_{\log n} \)
- height of that tree is \( \log n \)

Binomial Heap: Union

How can we merge two binomial tree heaps of the same size (\( 2^k \))?

- connect roots of \( H' \) and \( H'' \)
- choose smaller key to be root of \( H \)

Runtime? \( O(1) \)

Binomial Heap: Union

Go through each tree size starting at 0 and merge as we go
Binomial Heap: Union

Analogous to binary addition

Running time?
- Proportional to number of trees in root lists: \(2^{O(\log N)}\)
- \(O(\log N)\)

19 + 7 = 26

Binomial Heap: Delete Min/Max

We can find the min/max in \(O(\log n)\).

How can we extract it?

Hint: \(B_k\) consists of binomial trees: \(B_{k_1}, B_{k_2}, ..., B_0\)
**Binomial Heap: Delete Min**

Delete node with minimum key in binomial heap \( H \).
- Find root \( x \) with min key in root list of \( H \), and delete
- \( H' \) — broken binomial trees
- \( H \leftarrow \text{Union}(H', H) \)

Running time? \( O(\log N) \)

---

**Heaps**

Similar to binomial heap
- A Fibonacci heap consists of a sequence of heaps
  - More flexible
  - Heaps do not have to be binomial trees
  - More complicated

(adapted from Figure 19.1, pg. 466)
Should you always use a Fibonacci heap?

- Extract-Max and Delete are $O(n)$ worst case
- Constants can be large on some of the operations
- Complicated to implement

Can we do better?