Disjoint Sets: Naive Implementations

Disjoint sets

A disjoint-set data structure supports the following operations:

- `MakeSet(x)` creates a singleton set $\{x\}$
- `Find(x)` returns ID of the set containing x:
  - if x and y lie in the same set, then `Find(x)==Find(y)`.
  - Otherwise, `Find(x)!=Find(y)`.
- `Union(x, y)` merges two sets containing x and y.

```
Preprocess(maze):
for each cell c in maze:
    MakeSet(c)
for each cell c in maze:
    for each neighbor n of c:
        Union(c, n)
```

```
IsReachable(A, B):
return Find(A) == Find(B)
```

```
MakeSet(i): # Running time O(1)
smallest[i] <- i # Use array smallest[1...n] where smallest[i] stores
the smallest element in the set to which i belongs, assuming we have n
objects labelled with integers.
```

```
Find(i): # Running time O(1)
return smallest[i]
```

```
Union(i, j): # Running time O(n)
i_id <- Find(i)
j_id <- Find(j)
if i_id == j_id:
    return
m <- min(i_id, j_id)
for k from 1 to n:
    if smallest[k] in {i_id, j_id}: # If we merge i and j, all elements
in smallest with same value as smallest[i] or smallest[j] are replaced
by min{i_id, j_id}
        smallest[k] <- m
```

Disjoint Sets: Efficient Implementations

Idea

- Represent each set as a rooted tree
- ID of a set is the root of the tree
- Use array `parent[1...n]`: `parent[i]` is the parent of `i`, or `i` if it is the root

```
MakeSet(i): # Running time: O(1)
parent[i] <- i
```

```
Find(i): # O(tree height)
while i != parent[i]:
    i <- parent[i]
return i
```

Tree merging

- Hang one of the trees under the root of another one
- To make the trees shallow, hang the shorter one

Union by rank heuristic

To quickly find the height of a tree, we will keep the height of each subtree in an array `rank[1...n]`: `rank[i]` is the height of the subtree whose root is `i`.

```
MakeSet(i):
parent[i] <- i
rank[i] <- 0
```

```
Find(i): # O(log n)
while i != parent[i]:
    i <- parent[i]
return i
```

```
Union(i, j): # O(log n)
i_id <- Find(i)
j_id <- Find(j)
if i_id == j_id:
    return
if rank[i_id] > rank[j_id]:
    parent[j_id] <- i_id
else:
    parent[i_id] <- j_id:
    if rank[i_id] == rank[j_id]:
        rank[j_id] <- rank[j_id] + 1
```

Remark: For any node i, `rank[i]` is equal to the height of the tree rooted at i.

Lemma

The height of any tree in the forest is at most $\log_2 n$.

Lemma

Any tree of height k in the forest has at least 2^k nodes.

Proof:

Induction on k.

- Base: Initially, a tree has height 0 and one node: $2^0 = 1$
- Step: A tree of height k results from merging two trees of height $k = 1$. By induction hypothesis, each of two trees has at least 2^{k-1} nodes, hence the resulting tree contains at least 2^k nodes.

Path compression

```
Find(i):
if i != parent[i]:
    parent[i] <- Find(parent[i])
return parent[i]
```

Iterated logarithm

The iterated logarithm of n, $\log^* n$, is the number of times the logarithm function needs to be applied to n before the result is less than or equal than 1:

$$\log^* n = \begin{cases} 0 & \text{if } n \leq 1 \\ 1 + \log^*(\log(n)) & \text{if } n > 1 \end{cases}$$

Example

n	$\log^* n$
$n = 1$	0
$n = 2$	1
$n \in \{3, 4\}$	2
$n \in \{5, 6, \ldots, 16\}$	3
$n \in \{17, \ldots, 65536\}$	4
$n \in \{65537, \ldots, 2^{65536}\}$	5

Lemma

Assume that initially the data structure is empty. We make a sequence of m operations including n calls to `MakeSet`. Then the total running time is $O(m \log^* n)$. The amortized time of a single operation is $O(\log^* n)$ which is nearly constant.

Priority Queues: Introduction

Priority queue

A priority queue is an abstract data type supporting the following main operations:

- `Insert(p)` adds a new element with priority `p`
- `ExtractMax()` extracts an element with maximum priority

Additional operations

- `Remove(it)` removes an element pointed by an iterator `it`
- `GetMax()` returns an element with maximum priority (without changing the set of elements)
- `ChangePriority(it, p)` changes the priority of an element pointed by `it` to `p`

Priority Queues: Binary Heaps

Binary max-heap

Binary max-heap is a binary tree (each node has 0, 1 or 2 children) where the value of each node is at least the values of its children. In other words, for each edge of the tree, the value of the parent is at least the value of the child (Heap property).

Basic operations

API	Descriptions
`GetMax()`	Return the root value Running time: $O(1)$
`Insert(p)`	Attach a new node to any leaf, and let the new node sifts up
`SiftUp(i)`	Swap the problematic node with its parent until the heap property is satisfied Running time: $O(\text{tree height})$
`ExtractMax()`	Replace root with any leaf, and let the new node sifts down
`SiftDown(i)`	Swap the problematic node with its larger children until the heap property is satisfied Running time: $O(\text{tree height})$
`ChangePriority(i, p)`	Change the priority and let the changed element sifts up or down depending on whether its priority

API	Descriptions
	decreased or increased Running time: $O(\text{tree height})$
`Remove(i)`	Change the priority of the element to ∞, let it sifts up and then extract maximum Running time: $O(\text{tree height})$

Complete binary tree

A binary tree is complete if all its levels are filled except possibly the last one which is filled from left to right.

Lemma

A complete binary tree with n nodes has height at most $O(\log n)$.

Proof:

- Complete the last level to get a full binary tree on $n' \geq n$ nodes and the same number of levels l
- Note that $n' \leq 2n$
- Then, $n' = 2^l - 1$ and hence $l = \log_2(n'+1) \leq \log_2(2n+1) = O(\log n)$

Remark: n is the number of nodes of complete binary tree with l levels. Let n' be the number of nodes to be filled in order to get a full binary tree with l levels. So, $n' \geq n$. Total number of nodes up to level $l-1$ is $1 + 2^1 + 2^2 + \ldots + 2^{l-2} = \frac{2^{l-1}-1}{2-1} = 2^{l-1} - 1$. If there are more than $n+1$ nodes at the last layer, than $n' = (n+1) + n = 2n + 1 > 2n$ and we need at least $\frac{n+1}{2}$ points at $l-1$ levels. Note that the total number of nodes up to level $l-1$ is strictly less than n by definition. However,

$$2^{l-1} - 1 < n$$
$$2^{l-2} < \frac{n+1}{2}$$

we can never have the number of node at $l-1$ level more or equal than $\frac{n+1}{2}$. For the last step, the number of nodes in a complete binary tree with n nodes with l levels is $n' = 1 + 2^1 + 2^2 + \ldots + 2^{l-1} = \frac{2^l - 1}{2-1} = 2^l - 1$.

Store complete binary tree as array

- `parent(i) = floor(i/2)`
- `leftchild(1) = 2i`
- `rightchild(i) = 2i + 1`

Keeping the tree complete

- To insert an element, insert it as a leaf in the leftmost vacant position in the last level, and let it sifts up.

- To extract the maximum value, replace the root by the last leaf, and let it sifts down.

```
Parent(i):
return floor(i/2)
```

```
LeftChild(i):
return 2*i
```

```
RightChild(i):
return 2*i + 1
```

```
SiftUp(i):
while i > 1 and H[Parent(i)] < H[i]:
    swap H[Parent(i)] and H[i]
    i <- Parent(i)
```

```
SiftDown(i):
maxIndex <- i
l <- LeftChild(i)
if l <= size and H[l] > H[maxIndex]:
    maxIndex <- l
r <- RightChild(i)
if r <= size and H[r] > H[maxIndex]:
    maxIndex <- r
if i != maxIndex:
    swap H[i] and H[maxIndex]
    SiftDown(maxIndex)
```

```
Insert(p):
if size == MaxSize:
    return ERROR
size <- size + 1
H[size] <- p
SiftUp(size)
```

```
ExtractMax():
result <- H[1]
H[1] <- H[size]
size <- size - 1
SiftDown(1)
return result
```

```
Remove(i):
H[i] <- infinity
SiftUp(i)
ExtractMax()
```

```
ChangePriority(i, p):
oldp <- H[i]
H[i] <- p
if p > oldp:
    SiftUp(i)
else:
    SiftDown(i)
```

Heap Sort

```
HeapSort(A[1...n]): # Not in-place, O(nlogn)
create an empty priority queue
for i from 1 to n:
    Insert(A[i])
for i from n downto 1:
    A[i] <- ExtractMax()
```

```
BuildHeap(A[1...n]): # O(nlogn)
size <- n
for i from floor(n/2) downto 1: # Start repairing the heap property in
all subtrees of depth 1, so start at n/2
    SiftDown(i)
```

```
HeapSort(A[1...n]): # in-place, O(nlogn)
BuildHeap(A)
repeat (n - 1) times:
    swap A[1] and A[size] # Put the largest element at the end
    size <- size - 1
    SiftDown(1) # Recalculate the largest element in the partial array
```

Running time of `BuildHeap(A[1...n])`

Level	# nodes	$T(\text{SiftDown})$
1	1	$\log_2 n$
2	2	
	...	

Level	# nodes	$T(\text{SiftDown})$
$l-1$	$\leq \frac{n}{4}$	2
l	$\leq \frac{n}{2}$	1

$$T(\text{BuildHeap}) \leq \frac{n}{2} \cdot 1 + \frac{n}{4} \cdot 2 + \ldots \leq n \cdot \sum_{i=1}^{\infty} \frac{i}{2^i} = 2n$$

$$\frac{1}{2} + \frac{2}{4} + \frac{3}{8} + \frac{4}{16} + \ldots = \frac{1}{2} + (\frac{1}{4} + \frac{1}{4}) + (\frac{1}{8} + \frac{1}{8} + \frac{1}{8}) + \ldots$$

Since the series converges absolutely and we can rearrange the terms:

$$(\frac{1}{2} + \frac{1}{4} + \frac{1}{8} + \ldots) + \frac{1}{2}(\frac{1}{2} + \frac{1}{4} + \frac{1}{8} + \ldots) + \ldots \frac{1}{4}(\frac{1}{2} + \frac{1}{4} + \frac{1}{8} + \ldots) = 1 + \frac{1}{2} + \frac{1}{4} + \ldots = 2$$

Partial sorting

Input: An array `A[1...n]`, an integer $1 \leq k \leq n$.

Output: The last k elements of a sorted version of A.

Remark: It can be solved in $O(n)$ if $k = O(\frac{n}{\log n})$

```
PartialSorting(A[1...n], k): # Running time O(n + klogn)
BuildHeap(A)
for i from 1 to k:
    ExtractMax()
```

0-based Arrays

```
Parent(i):
return floor((i - 1)/2)
```

```
LeftChild(i):
return 2*i + 1
```

```
RightChild(i):
return 2*i + 2
```

Binary min-heap

Binary min-heap is a binary tree (each node has 0, 1 or 2 children) where the value of each node is at most the values of its children.

d-ary Heap

- In a d-ary heap, nodes on all levels except for possibly the last one have exactly d children
- The height of such a tree is about $\log_d n$
- The running time of `SiftUp` is $O(\log_d n)$
- The running time of `SiftDown` is $O(d \log_d n)$: on each level, we find the largest value among d children