Data Structures - Week 3

Tuesday, March 28, 2023 10:23 AM

Priority queues: Introduction

Queue

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

- PushBack(e) adds an element to the back of the queue
- PopFront() extracts an element from the front of the queue

Priority Queue

A priority queue is a generalization of a queue where each element is assigned a priority and elements come out in order by priority.

Typical use cases

Scheduling jobs:

- Want to process jobs one by one in order of decreasing priority. While the current job is processed, new job may arrive.
- To add a job to the set of scheduled jobs, call Insert(job)
- To process a job with the highest priority, get it by calling ExtractMax()

Priority Queue (formally)

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 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

Algorithms that use priority queues

- Dijsktra's algorithm: finding a shortest path in a graph
- Prim's algorithm: constructing an optimum prefix-fre encoding of a string
- Head sort: sorting a given sequence

<u>Priority queues: Binary Heaps</u>

Binary max-heap:

- A binary tree (each node has zero, one, or two children) where the value of each note is a at least the values of its children
- For each edge of the tree, the value of the parent is at least the value of the child

GetMax

- Return the root value
- Running time O(1)

Insert

- Attach a new node to any leaf
- This may violate the heap property
- To fix this we let the new node sift up

SiftUp

- For this, we swap the problematic node with its parent until the property is satisfied
- Invariant: heap property is violated on at most one edge
- The edge gets closer to the root while sifting up
- Running time O(tree height)

ExtractMax

- Replace the root with any leaf
- Again this may violate the heap property
- To fix it, we let the problematic node sift down

SiftDown

SiftDown

- For this we swap the prolematic node with larger child until the heap property is satisfied
- We swap with the larger child which automatically fixes one of the two bad edges
- Running time: O(tree height)

ChangePriority

- Change the priority and let the changed element sift up or down depending on whether its priority decreased or increased
- Running time: O(tree height)

Remove

- Change the priority if the element to infinite, let it sift up and the extract the maximum
- Now call ExtractMax()
- Running time: O(tree height)

How to keep a tree shallow?

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

First Advantage: Low Height

- A complete binary tree with n nodes has a height at most O(log n)

- What do we pay for these advantages?
- We need to keep the tree complete
- Which binary heap operations modify the shape of the tree?
- Only Insert and ExtractMax(remove changes the chape by calling ExtractMax)

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 sift up
- To extract the maximum value replace the root by the last leaf and let it sift down

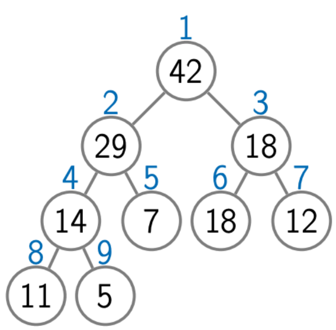
General Setting

- maxSize is the maximum number of elements in the heap
- Size is the size of the heap
- H[1…maxSize] is an array of length maxSize where the heap occupies the first size elements

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)
```

Example



$size = 9$

$maxSize = 13$

	1	2	3	4	5	6	7	8	9	10	11	12	13
H	42	29	18	14	7	18	12	11	5	30	29	2	8

ExtractMax()

```
result ← H[1]
H[1] ← H[size]
size ← size − 1
SiftDown(1)
return result
```

Insert(p)

```
if size = maxSize:
    return ERROR
size ← size + 1
H[size] ← p
SiftUp(size)
```

Remove(i)

```
H[i] ← ∞
SiftUp(i)
ExtractMax()
```

SiftUp(i)

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

ChangePriority(i, p)

```
oldp ← H[i]
H[i] ← p
if p > oldp:
    SiftUp(i)
else:
    SiftDown(i)
```

Disjoint Sets: Naïve Implementations

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 the Find(x) = Find(y)
    - Otherwise, Find(x) != Find(y)
- Union(x, y) merge 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)
```

Naïve Implementations

For simplicity, we assume that our n objects are just integers 1,2,…,n

Using the smallest element as ID

- Use the smallest element of the set as its ID
- Use array smallest[1…n]: smallest[i] stores the smallest element in the set I belongs to

belongs to

MakeSet(i)

```
smallest[i] ← i
```

Find(i)

```
return smallest[i]
```

Running time: $O(1)$

Union(i, j)

```
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}:
    smallest[k] ← m
```

Running time: $O(n)$

- Current bottleneck: Union
- What basic data structure allows for efficient merging?
- Linked list!
- Idea: represent a set as a linked list, use the list tail as ID of the set

- Pros:
    - Running time of Union is O(1)
    - Well defined ID
- Cons:
    - Running time of Find is O(n) as we need to traverse the list to find its tail
    - Union(x, y) works in time O(1) only if we can get the tail of the list of x and the head of the list of y in constant time!

Disjoint Sets: effcient implementations

- 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)

MakeSet(i)

parent[i] $\leftarrow i$

Running time: $O(1)$

Find(i)

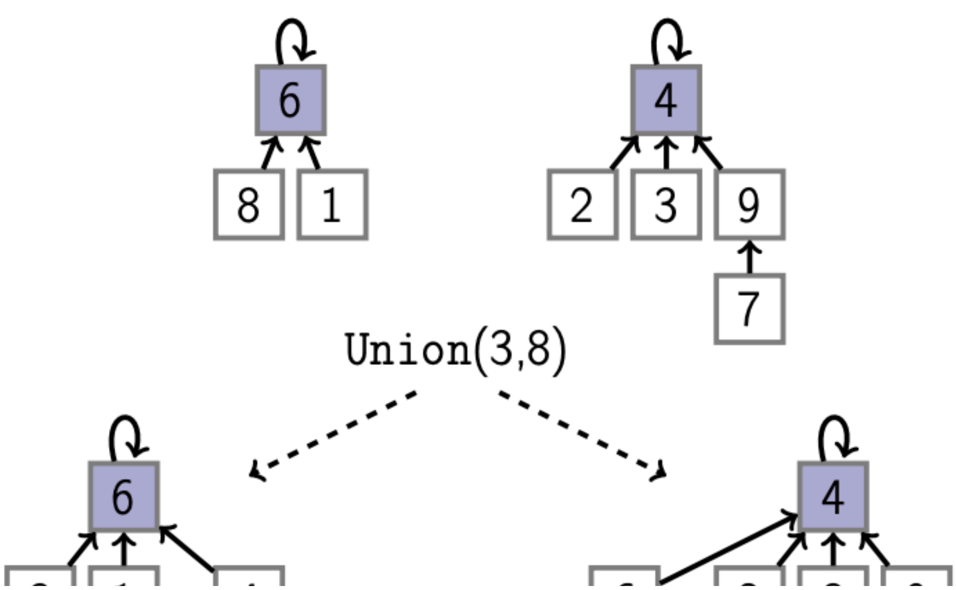
while $i \neq$ parent[i]:

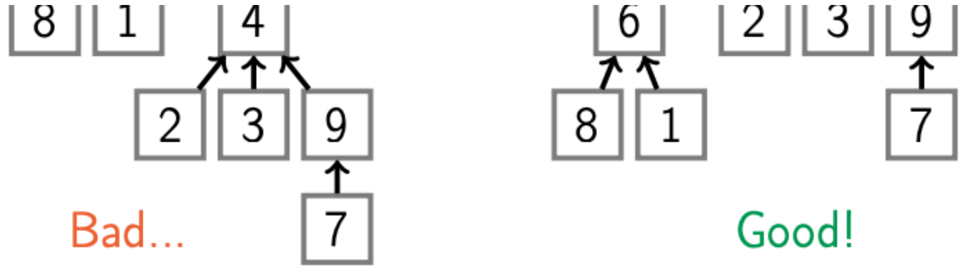
  $i \leftarrow$ parent[i]

return i

Running time: $O(\text{tree height})$

- How to merge two trees?
  - Hang one of the trees under the root of the other one
- Which one to hang?
  - A shorter one, since we would like to keep the trees shallow





- When merging two trees we hang a shorter one under the root of a taller one
- To quickly find a 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
- (The reason we call it rank, but not height will become clear later)
- Hanging a shorter tree under a taller one is called a union by rank heuristic