Lösningsförslag till tentamen

Datastrukturer, DAT037, 2018-08-23

1. Loopen upprepas `n` gånger. `q.dequeue` tar $O(1)$. `s.contains` tar $O(\log(2\mathtt{n})) = O(\log \mathtt{n})$. `l.addLast` tar $O(1)$ (amorterat). `l.addFirst` tar $O(\mathtt{i}) = O(\mathtt{n})$.

   Kodens tidskomplexitet är

   $$O(n(1 + \log \mathtt{n} + \max(1, \mathtt{n}))) = O(n^2)$$

2. Nedan betyder ett streck att ingen väg till noden ännu är känd och fetstil betyder att närmsta avståndet har hittats.

A	*B*	*C*	*D*	*E*	*F*	*G*	*H*	*I*
0	-	-	-	-	-	-	-	-
0	5	-	3	-	-	-	-	-
0	5	-	**3**	9	-	4	-	-
0	5	-	**3**	9	-	**4**	6	-
0	**5**	10	**3**	8	-	**4**	6	-
0	**5**	10	**3**	7	-	**4**	**6**	8
0	**5**	10	**3**	**7**	10	**4**	**6**	8
0	**5**	10	**3**	**7**	9	**4**	**6**	**8**
0	**5**	10	**3**	**7**	**9**	**4**	**6**	**8**
0	**5**	**10**	**3**	**7**	**9**	**4**	**6**	**8**

3. För trea kan man sätta in elementen sorterat i en dynamisk array. `add`, `remove` och `member` tar då $O(n)$. `ithsmallest` implementeras med vanlig array-indexering och tar $O(1)$.

   För fyra eller femma kan man utgå från ett balanserat sökträd, t ex ett AVL-träd. Lägg till ett nytt fält i varje nod, ett heltal som lagrar vänstra delträdets storlek.

   `member` påverkas inte av ändringen. $O(\log n)$.

   Insättningen i `add` kan implementeras så här:

```
add(x) =
  newroot = add_rec(x, root)
  if newroot != null then {
    root = newroot
  }
```

```
add_rec(x, node) =
  if node == null then {
    newnode = new Node
    newnode.contents = x
    newnode.leftsize = 0
    return newnode
  } else {
    if x == node.const then {
      return null
    } else if x < node.const then {
      newnode = add_rec(x, node.left)
      if newnode == null then {
        return null
      } else {
        node.left = newnode
        node.leftsize = node.leftsize + 1
        return node
      }
    } else {
      newnode = add_rec(x, node.right)
      if newnode == null then {
        return null
      } else {
        node.right = newnode
        return node
      }
    }
  }
```

Skillnaden jämfört med insättning i vanligt balanserat träd tar bara konstant tid per nod. Ombalanseringen kan brytas ned i enkla vänster- och högerrotationer. I varje rotation måste storleksfältet i en nod ändras. Detta tar konstant tid per rotation. Ombalanseringen och därmed hela `add` tar alltså $O(\log n)$.

På motsvarande sätt kan `remove` modifieras utan att komplexiteten påverkas.

Slutligen kan `ithsmallest` implementeras så här:

```
ithsmallest(i) =
  node = root
  while (i != node.leftsize) {
    if i < node.leftsize then {
      node = node.left
    } else {
      node = node.right
      i = i - 1 - node.leftsize
    }
  }
  return node.contents
```

Loopen upprepas en gång per nivå i trädet, så operationen tar $O(\log n)$.

4. *För trea:*

2	5	3	10	8	13	8	12	13	15	20	20	14	16

För fyra eller femma:

Se kurslitteraturen.

5. Se kurslitteraturen.

6. *För trea:*

Man kan lagra medlemstalparen i en dynamisk array och mittpunkten som två heltalsfält. `setcenter` ändrar desaa två fält. $O(1)$. `add` lägger till ett talpar i slutet på arrayen. $O(1)$ amorterat. `member` letar igenom alla talpar. $O(n)$. `removeclosest` går igenom alla talpar och beräknar avståndet till mittpunkten. Tar bort en av de med minst avstånd. $O(n)$.

För fyra eller femma:

Man kan lagra talparen i två strukturer, en hashtabell, `s`, och en binär min-heap, `h`. Två heltalsfält, `mittx` och `mitty`, lagrar mittpunkten. I heapen använder man avståndet till mittpunkten som prioritet. I hashtabellen använder man en hashfunktion som kombinerar hashkoden för de båda talen i talparet. När mittpunkten ändras bygger man en helt ny min-heap. Det tar $O(n)$.

```
klass Talpar {
 int x, y
}

empty() =  // O(1)
  s = tom hashtabell med talpar som element och
       hashfunktion = hash(x, y)
  h = tom min-heap med talpar som element och
       komparator = abs(x - mittx) + abs(y - mitty)
  mittx = 0
  mitty = 0

setcenter(x, y) =  // O(n)
  mittx = x
  mitty = y
  h = build-heap från gamla h  // O(n)

add(x, y) =  // O(log n)
  s.add(Talpar(x, y))  // O(1)
  h.add(Talpar(x, y))  // O(log n)

member(x, y) =  // O(1)
  return s.member(Talpar(x, y))  // O(1)

removeclosest() =  // O(log n)
  if h.isempty() then return  // O(1)
  Talpar xy = h.delete-min()  // O(log n)
  s.remove(xy)  // O(1)
```