TDDB56 DALGOPT – Algoritmer och Optimering

Kontrollskrivningen 2006-10-19, 8–13

Outline of the proposed solutions for selected problems.

1. (3 p)

(a) The justification is not included. See similar problems in lektionerna.

   - false
   - true
   - true
   - false

(b) As \( f \in O(g) \), then there exist constants \( c_f > 0 \) and \( n_0, n_f \geq 0 \) such that for every \( n \geq n_f \) we have \( f(n) \leq c_f g(n) \). Similarly, we have constants \( c_g \) and \( n_0, n_g \), such that for every \( n \geq n_g \) we have \( g(n) \leq c_g h(n) \). Take \( n_0 = \max(n_0, n_f, n_g) \) and \( n \geq n_0 \). Then \( f(n) \leq c_f g(n) \leq c_f c_g h(n) \). Hence \( f \in O(h) \) and the respective constants are \( c = c_f c_g \) and \( n_0 \) as defined above.

2. (3 p)

(a) • MakeEmptyQueue create two empty stacks: \( E \) and \( D \).

   • To enqueue data item \( d \) push it on \( E \).

   • To dequeue:

     If \( D \) is empty and \( E \) is empty report error.

     Otherwise:

     i. If \( D \) is empty, pop one-by-one all elements of \( E \) and push them on \( D \). After all this enqueued data items are placed on \( D \) from top to bottom in order of enqueuing: the most recently enqueued one is in the bottom and the Front element is placed on the top of \( D \). We pop it to complete the dequeue operation.

     ii. Otherwise pop from \( D \).

     • Front operation is as dequeue but the data item is not popped from \( D \).

     • IsEmptyQueue is implemented as two checks: IsEmptyStack(\( E \)) and IsEmptyStack(\( D \)); it returns true if both checks return true.

   The pseudocode not included.

(b) We assume that each stack operation runs in \( \Theta(1) \).

   i. Enqueue operation is always a push on \( E \), so it runs in constant time \( \Theta(1) \).

   ii. • A single enqueue operation is implemented as push on \( E \), thus works always in \( \Theta(1) \); there is no specific worst case.
• The worst case of dequeue is if it is requested after pushing $n$ data items on $E$. In that case we need:
  (1) emptiness check of both stacks done in $\Theta(1)$
  (2) $n$ pop operations and $n$ emptiness checks on $E$ done in $\Theta(n)$
  (3) $n$ push operations on $D$ done in $\Theta(n)$
  (4) one pop operation on $D$ done in $\Theta(1)$
Thus the worst case dequeue takes $\Theta(n)$ time.

• Whatever is the sequence of the operations every data item is exactly once pushed on $E$, popped from $E$, pushed on $D$ and popped from $D$. Thus the amortized execution time needed to enqueue and dequeue $n$ data items is $O(n)$.

3. (3 p)
   (a) The hash table:

   Before removal:
   
   After all operations:

   (b) The skip list:

   Before removal:

   After all operations:
4. (3 p)
(a) •

The tree

```
/   \
41(-1) 16(-1) 67(1)
   /\   /\   /\  
  39 49(0) 74 47 61
```

is an AVL tree, since the balance of each node is 0, 1 or -1.

• (1.5)

Insert(40) final result:

```
T1
   41
    /\ 
   39 67
    /\  /\  
  16 40 49 74
    / \  
   47 61
```

Insert(53) The critical node is 67. Insertion will include 53 as the left child of 61. After that re-balancing is needed. It is achieved by two rotations: left rotation around 49:

```
/   \
41 16 67
    /\  /\ 
  39 61 74
    / \ 
   49 
    / \ 
   47 53
```

and right rotation around 67. Final result:

```
T2
   41
    /\ 
   16 61
    /\  /\  
  39 49 67
    / \  
   47 53 74
```
Delete(67) First step replaces 67 by its inorder successor 74:

```
  41 (-1)
   /  \
(-1)16  74 (2)
   \  /
    39 49 (0)
   /  \
   47  61
```

This is not an AVL tree; the balance of 74 is to be restored by right rotation. The obtained tree is an AVL tree.

```
  41 (-1)
   /  \
(-1)16  49 (-1)
   \  /
    39 47 74 (1)
   /  \
   61
```

An alternative solution is to replace 67 by its inorder predecessor 61, in which case no rebalancing is needed.

The above solution shows application of each the operations to the original tree. If each of the operation is applied to the result of the previous one then: (1) T2' obtained by insertion of 53 to T1 has the left subtree as in T1 and right subtree as in T2; (2) Removal of 67 from T2' does not destroy the AVL property and does not require any re-balancing.

(b)

```
41
  16 41

39
 /  \
16 41

39
 /  \
16 41 67

39 49
 /  |  \
16 41 67

39 49
 /  |  \
16 41 47 67
```
5. (4 p)

(a) Insertion Sort (1)

7 5 9 6 8
5 7 9 6 8
5 6 7 9 8
5 6 7 8 9

Insertion Sort (textbook p.104) is stable since in the moment of insertion into the sorted part the inserted element is strictly smaller than all following elements in the sorted part. Thus the relative order of two elements with equal keys is preserved at every step of sorting.

(b) Quick Sort (1)

7 5 9 6 8  partition starts: pivot 8
7 5 6 9 8
7 5 6 8 9  end of partition
7 5 6 8 9
7 5 6 8 9  partition of 7 5 6 starts: pivot 6
5 7 6 8 9
5 6 7 8 9

(c) Heap Sort; the trees are not shown (2)

7 5 9 6 8  Start of heapification
9 5 7 6 8
9 8 7 6 5  max-heap created
5 8 7 6 9  maximal element moved from the heap
8 5 7 6 9  re-heapification of the prefix
8 6 7 5 9
5 6 7 8 9  maximal element moved from the heap
7 6 5 8 9  re-heapification
5 6 7 8 9  maximal element moved from the heap
5 7 8 9  re-heapification
5 6 7 8 9

6. (4 p)

(a) The following graph shows the restrictions: (0.5)
(b) We assume here that the neighbours of a vertex are explored in the order given by the task numbering in the problem. DFS: 1, 2, 3, 7, 6, 12, 8, 11, 5, 10, 9, 4

(c) BFS: 1, 2, 4, 3, 5, 9, 7, 8, 10, 11, 6, 12

(d) This is a directed acyclic graph and the problem can be solved by the topological sort algorithm based on the depth-first search (see the slides). Another topological algorithm, based on removing edges without incoming vertices is discussed in p. 617 of the course book.

The DFS-based algorithm in the case of the DFS shown above assigns the following numbers to the tasks:

- Moving in: 12
- Painting interior: 11
- Floor: 10
- Insulation: 9
- Doors: 8
- Interior Walls: 7
- Roof cover: 6
- Roof: 5
- Windows: 4
- Exterior Walls: 3
- Chimney: 2
- Ground: 1

The schedule is determined by undertaking the task in increasing order of their computed numbers. The schedule is not unique: a different principle of selecting the adjacent nodes during DFS may result in a different schedule, e.g:

- Moving in: 12
- Painting interior: 11
- Floor: 10
- Roof cover: 9
- Roof: 8
- Chimney: 7
- Insulation: 6
- Doors: 5
- Interior walls: 4
- Windows: 3
- Exterior walls: 2
- Ground: 1