The University of Newcastle
School of Electrical Engineering and Computer Science

COMP2230/COMP6230 Algorithms

Tutorial Week 4 Solutions

9th – 13th August 2021

Tutorial:

1. Write a function *invert(s)* that inverts the contents of a stack. For example, if stack s originally contained 1, 2 and 3, where 3 is the most recently added item, after *invert(s)* s contains 3, 2 and 1, where 1 is the most recently added item. Use the abstract data type stack (do not refer to *data* or *t*). You may use additional stacks.

   Solution:

```
invert(s) {
   \\s1 and s2 are auxiliary stacks
   s1.stack_init()
   s2.stack_init()
   while (!s.empty()){
         s1.push(s.top())
         s.pop()
   }
   while (!s1.empty()){
         s2.push(s1.top())
         s1.pop()
   }
   while (!s2.empty()){
         s.push(s2.top())
         s2.pop()
   }
}
```

2. Suppose that *start* is a reference to the first node in the linked list or, if the linked list is empty, *start* = null. Write an algorithm that is passed *start* and a value *val*. The algorithm adds a node at the beginning of the linked list whose data is *val* and returns start. What is the worst case time of your algorithm?

   Solution:

```
add_at_beginning (start, val) {
      temp = new node
      temp.data = val
      temp.next = start
      start = temp
      return start
}
```

   Worst-case time is $\Theta(1)$.

3. A priority queue is implemented using an array. An item is inserted by putting it at

the end of the array. Show that, in the worst case, performing n insertions and n deletions in an initially empty priority queue takes time $\Theta(n^2)$.

Solution:
Suppose that the values $n, n-1, n-2, \ldots, 1$ are inserted in that order. Then the n deletions are performed in the same order; that is, n is deleted first, then $n-1$, and so on. Since n is the first element in the array, $n-1, n-2, \ldots, 1$ must each move one cell, which takes at least $n-1$ steps. When the second deletion occurs, $n-1$ is deleted. Since it is now the first element in the array, $n-2, n-3 \ldots, 1$ must each move one cell, which takes at least $n-2$ steps, and so on. Therefore, the total time is at least $(n-1)+(n-2)+\ldots+1 = (n-1)n/2 = \Theta(n^2)$.

4. Write a linear time algorithm that swaps the left and right children of each node in a binary tree.

Solution:

```
swap(root){
if (root == null) return;
swap(root.left);
swap(root.right);
temp = root.left;
root.left = root.right;
root.right = temp;
}
```

5. Write a non-recursive version of Algorithm 3.4.14 (inserting in a binary search tree) from the textbook.

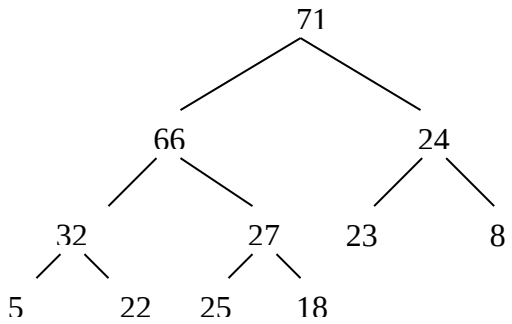
```
BSTinsert(root,val){
      //set up node to be added to tree
      temp=new node
      temp.data=val
      temp.left=temp.right=null
      if(root==null) return temp
      BSTinsert_recurs(root,temp)
      Return root
}

BSTinsert_recurs(root,temp){
      If(temp.data ≤ root.data)
            If(root.left==null)
                  Root.left=temp
            Else
                  BSTinsert_recurs(root.left,temp)
      Else
            If(root.right==null)
                  Root.right=temp
            Else
                  BSTinsert_recurs(root.right,temp)
}
```

Solution: See textbook, p.669, 3.4.16.

```
BSTinsert(root,val){
//set up node to be added to tree
      temp=new node
      temp.data=val
      temp.left=temp.right=null
      if(root==null) return temp
      where=root
      while(true)
            if(val ≤ where.data)
                  if(where.left==null){
                        where.left=temp
                        return root
                  }
                  else
                        where=where.left
            else
                  if(where.right==null){
                        where.right=temp
                        return root
                  }
                  else
                        where=where.rightt
}
```

6. Trace insert when 61 is inserted in Figure 3.5.5(e).

```
Heap_insert(val,v,n){
      i=n=n+1
//i is the child i/2 parent
//if i >1, i is not the root
      while(i>1 && val >v[i/2]){
            v[i]=v[i/2]
            i=i/2
      }
      v[i]=val
}
```



Solution:

setup

71	66	24	32	27	23	8	5	22	25	18	

Val=61, i=12

Val>v[6], so move v[6] to v[12], i=i/2

71	66	24	32	27	23	8	5	22	25	18	23

Val=61, i=6

Val>v[3], so move v[3] to v[6], i=i/2

71	66	24	32	27	24	8	5	22	25	18	23

Val=61, i=3

Val<v[1], so insert val into v[3],i=i/2

71	66	61	32	27	24	8	5	22	25	18	23

Val=61, i=3

7. Assuming the data in the heap are distinct, what are possible locations for:

   - The smallest element?

   Solution: The smallest element in a heap may only be at a leaf node.

   - The second smallest element?

   Solution: The second smallest element may be at a leaf, or at a node with only one child (specifically the smallest element). Note that there is only ever at most one node with one child at any given time.

   - The second largest element?

   Solution: The second largest element must be a child of the root.

8. Show that inserting *n* elements into an initially empty heap takes time Θ(*n* log *n*) in the worst case.

   Solution*:* The longest that any element must travel when inserted is from the insertion point to the root. The height of the heap with *k* vertices is $\lfloor log\ k \rfloor$. With *n* insertions, we get the formula:

$$T(n) = \sum_{i=1}^{n} \lfloor \lg i \rfloor \leq \sum_{i=1}^{n} \lg n = n \lg\ n, n \geq 1$$

   Therefore, $T(n) = O(n \lg n)$.

$$T(n) = \sum_{i=1}^{n} \lfloor \lg i \rfloor \geq \sum_{i=\lceil \frac{n}{2} \rceil}^{n} \lfloor \lg i \rfloor \geq \left\lceil \frac{n}{2} \right\rceil \left\lfloor \lg \left\lceil \frac{n}{2} \right\rceil \right\rfloor \geq \frac{n}{2} \left\lfloor \lg \frac{n}{2} \right\rfloor \geq \frac{n \lg n}{4}, n \geq 16$$

   as

$$\left\lfloor \lg \frac{n}{2} \right\rfloor \geq \lg \frac{n}{2} - 1 = \lg \frac{n}{4} \geq \frac{\lg n}{2}\ for\ n\ \geq 16$$

   Therefore, $T(n) = \Omega(n \lg n)$.

9. Given the *key* array:

18	24	71	5	22	23	66	32	8	25	27

Show the *into* and *outof* arrays when the heap in figure 3.5.5(e) of the text is implemented as an indirect heap

Solution:

Into:

11	3	1	8	9	6	2	4	7	10	5

Outof:

3	7	2	8	11	6	9	4	5	10	1

Workshop:

10. Write a version of enqueue that checks for full array. If the array is full, the functions simply returns false. If the array is not full, the behaviour is the same as original enqueue, except that the function also returns true.

Solution:

```
enqueue(val) {
      if ((r + 1) mod SIZE == f)
            return false
      if (empty())
            r = f = 0
      else {
            r = r + 1
            if ( r == SIZE )
                  r = 0
      }
      data[r] = val
      return true
}
```

11. Write an algorithm to reverse a linked list. You may modify only the next fields of the nodes. What is the worst case time of your algorithm?

Solution:

```
reverse(s) {
      \\b, t and a are three consecutive nodes in the original list
      b = null
      t = s
      while (t != null) {
            a = t.next
            t.next = b
            b = t
            t = a
      }
      return b
```

```
}
```

Worst-case time is Θ(n), where n is the number of nodes in the linked list.

12. Show how to implement a queue using priority queue.

Solution:
Maintain a global variable *priority*, which is intialised to 0. When an element is added to the queue, decrement priority, and then add the item to the priority queue with priority *priority*.

13. Prove that postorder runs in time Θ(n) when the input is an n-node binary tree.

Solution: Adapted from solutions to exercise 3.4.1, page 668 of Algorithms (Johnsonbaugh and Schaeffer, 2004). Note that the solution given in the textbook proves for inorder but it applies to postorder too.

We use induction to prove that if a binary tree has n nodes, the conditional line,

```
if (root != null)
```

is executed $2n + 1$ times.

The proof is by induction on n.
The basis step is $n = 0$. In this case, the conditional line is executed $1 = 2 \cdot 0 + 1$ time, which verifies the basis step.

Now assume that $n > 0$ and if a binary tree has $m < n$ nodes, the conditional line is executed $2m + 1$ times.

Let T be an n-node binary tree with root root. Suppose that T's left subtree contains n_l nodes and T's right subtree contains n_r nodes. When postorder is called with input *root*, the first line

```
if (root != null)
```

is executed, which accounts for one execution of the conditional line.
Next, postorder is called with input *root.left*. The tree rooted at root.left has $n_l < n$ nodes; so, by the inductive assumption, the conditional line is executed a total of $2n_l + 1$ times while processing T's left subtree. Similarly, the tree rooted at *root.right* has $n_r < n$ nodes; so by the inductive assumption, the conditional line is executed a total of $2n_r + 1$ times while processing T's right subtree. The total number of times the conditional line is executed is, therefore, $1 + (2n_l + 1) + (2n_r + 1) = 2(n_l + n_r + 1) + 1 = 2n + 1$.

The inductive step is complete.

14. Write an algorithm that returns the number of terminal nodes (leaves) in a binary tree.

Solution:

```
leaves(root){
      if (root == null) return 0;
      if (root.left == null && root.right == null) return 1;
      return (leaves(root.left) + leaves(root.right));
}
```

15. Write a linear time algorithm that determines whether a binary tree is a binary search tree.

Solution:

```
Input Parameters: root (assumed not null)
Output Parameters: minval,maxval

check_BST(root, minval, maxval){
      minvalL = ∞ //to avoid special cases
      maxvalR = -∞ //to avoid special cases
      if (root.left != null)
            if (check_BST(root.left, minvalL, maxvalL)==false ||
                maxvalL > root.data)
                  return false
      if (root.right != null)
            if (check_BST(root.right, minvalR, maxvalR)==false ||
                minvalR ≤ root.data)
                  return false
      minval=min(minvalL,root.data)
      maxval=max(maxvalR,root.data)
      return true
}
```

16. Write an algorithm to search for a particular item in a binary search tree. The worst case time must be Θ(h) where h is the height of the tree. If the item is found, return a reference to the location. If it is not found, return null.

Solution:

```
find(root,val){

      if (root == null) return null;
      if (root == val) return root;
            if (root < val) return find(root.right,val);
      return find(root.left,val);
      }
```

17. Write a constant time algorithm for returning the largest value in an indirect heap. Show that the algorithm runs in constant time.

Solution:

```
getMax(key, into, outof){
      return key[outof[1]];
}
```

Extra Questions:

18. Write an algorithm to merge two linked lists. Assume that the data in each list are in nonincreasing order. The result is one linked list, containing all of the data, in nonincreasing order. You may modify only the next fields of the nodes. What is the worst case time of your algorithm?

19. A deque (pronounced “deck”) is like a queue, except that items may be added and deleted at the rear or the front. Implement a deque using an array, and implement it using a linked list. Do not incorporate error checking into your functions.

20. Show how to implement a stack using priority queue.

21. A binary expression tree is a binary tree in which the internal nodes represent binary operators and the terminal nodes represent values. An operator operates on its left and right subtrees. Write a postorder algorithm that prints the data in a binary expression. (i.e. The algorithm produces an arithmetic expression in postfix notation.)

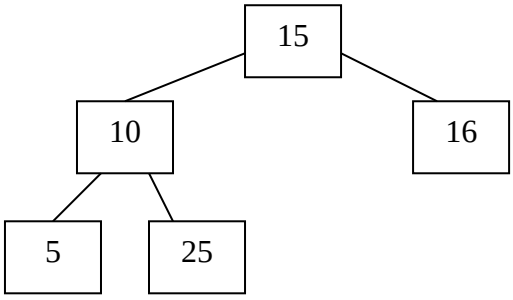
Solution:

```
rpn(root){
    if(root.left!=null)    {
      rpn(root.left)
      rpn(root.right)
    }
    print(root.data)
}
```

22. If T is a binary tree where at each node v the datum in v's left child is less than or equal to the datum in v, and the datum in v's right child is greater than or equal to the datum at v, then T is a binary search tree. Is this statement true or false? Explain.

Solution:

The definition above is not a definition of a binary search tree. To see this consider the following tree which would pass the definition although it is not a binary search tree:



Note the difference between “left child” and “left subtree”.

23. Write an O(log n) siftup algorithm for a heap. The input to siftup is an index i and a

heap structure in which the value of each node is greater than or equal to the values of its children (if any), except for the node at index i which may have a value greater than its parent. siftup is to restore the heap property. Prove the running time.

Solution:

```
Siftup(v,i){
      temp=v[i]
      while(i>1 && temp>v[i/2]){
            v[i]=v[i/2]
            i=i/2
      }
      v[i]=temp
}
```

The loop is executed $\lfloor lg\ i \rfloor$ times, where $i \leq n$. Thus the running time is $O(lg\ n)$.

24. Write an algorithm that inserts a value into an indirect heap. This algorithm should run in logarithmic time. Show that it does.

Solution:

We first insert the value *val* at the index $n+1$ in the key array, and then insert it into the heap.

```
 insert(val,n){
      i=n=n+1;
      key[n+1]=val
      //i is the child and i/2 is the parent.
      while(i>1 && val>key[outof[i/2]]){
            //copy the parent value to child
            outof[i]=outof[i/2]
            into[outof[i]]=i
            //move i to parent
            i=i/2;
      }
      //insert val at index i
      outof[i]=n+1
      into[n+1]=i
 }
```

In the worst case, the value added travels all the way from the last level to the root; thus the worst-case time is $\Theta(lg\ n)$.

25. Suppose that in heapsort (p. 145 in the text), the call to heapify (Algorithm 3.5.12) is replaced with repeated calls to insert (Algorithm 3.5.10) to create the initial heap. What is the worst case running time of this version of heapsort?

```
Heapsort(v,n){
      //make v into a heap
      heapify(v,n)
      for i=n downto 2{
            //v[1] is the largest among v[1],…,v[i]
            //put it in the correct cell
            swap(v[1],v[i]
            //heap is now at indexes 1 through i-1
            //restore heap

            siftdown(v,1,i-1)
      }}
```

```
heapify(v,n){
      //n/2 is the index of the parent of the last node
      for i=n/2 downto 1
            siftdown(v,i,n)
}
```

Solution: Θ(n lg n)