Final

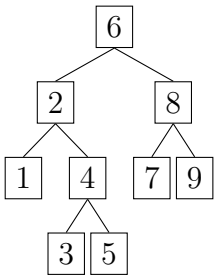
Note

This is a closed-book exam. Each problem accounts for 10 points, unless otherwise marked.

Problems

1. Consider the infix to postfix conversion algorithm discussed in class (see the appendix), which assumes all operators are associated to the left. Suppose we want to allow the unary `-` (negation), which has higher precedence than `*` and `/` and is associated to the right, i.e., `- - a` equals `- (- a)`. Please modify the conversion algorithm to allow the additional unary `-`. Is it necessary to use a new symbol to represent the unary `-` in the postfix expression? Please explain.

2. Below is a binary tree with 9 nodes, each storing an integer key value.



   For each of the following orders of traversal, write the sequence of nodes (represented by their key values) visited in that order.

   (a) preorder

   (b) postorder

3. This problem concerns the notion of "inorder successor" in a binary search tree.

   (a) If the inorder successor of a node with two children is an internal (or non-leaf) node, then that inorder successor must not have a left child. Why?

   (b) Give an algorithm (in suitable pseudocode) that finds the inorder successor of a given node.

4. Given a binary search tree, you may save the key values in a file using the preorder traversal and later restore the original tree by inserting the key values in the order saved into an empty tree.

   (a) Prove by induction (or give convincing arguments) that the statement above indeed is true.

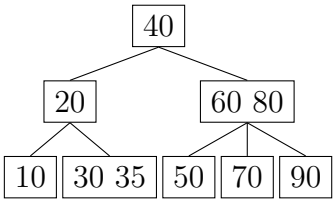
(b) Given a sequence of key values, if we insert the key values into an empty tree and then traverse the tree in preorder, will we get the original sequence of key values? Please justify your answer.

5. Rearrange the entries in the following array so that it becomes a (max) heap. You may proceed in either a "top-down" or a "bottom-up" fashion. Show the contents of the intermediate array after an entry is added to the processed region.

1	2	3	4	5	6	7	8	9	10	11	12
6	2	4	8	11	1	5	7	9	10	3	12

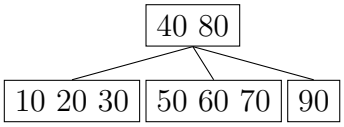
6. Describe the two main approaches to resolving collisions in hashing. Please be succinct, yet to the point.

7. Below is a 2-3 tree with 8 nodes, storing 10 key values totally.



Suppose we now want to delete 20, 70, 80, and 90 (in this order) from the tree. Please show the resulting tree after the completion of each deletion.

8. Below is a 2-3-4 tree with 4 nodes, storing 9 key values totally.



Suppose we now want to insert 85, 55, and 25 (in this order) into the tree. Please show the resulting tree after the completion of each insertion. (Note: Recall that the insertion algorithm for a 2-3-4 tree splits every 4-node encountered on the way down the tree from the root to the leaf where the input data item is to be placed.)

9. Consider the 2-3-4 tree again in the preceding problem. Give a sequence of key values such that insertions of the key values result in the tree as shown above. Please show the intermediate tree after the insertion of each key value.

10. Below is the algorithm we discussed in class for the BFS traversal of a graph.

```
bfs(v: Vertex)
   q = a new empty queue
   q.enqueue(v)
   Mark v as visited
```

```
while (!q.isEmpty()) {
  w = q.peekFront()
  q.dequeue()

  for (each unvisited vertex u adjacent to w) {
    Mark u as visited
    q.enqueue(u)
  }
}
```

(a) Modify the code to use a stack so that it becomes an algorithm for the DFS traversal of a graph. Be careful about when the marking of an unvisited node should be done.

(b) Modify the code further so that each node is assigned a DFS number.

Appendix

- Below is the algorithm discussed in class for converting an infix expression to a postfix expression.

```
for (each character ch in the infix expression) {
  switch (ch) {
    case operand:  // Append operand to the end of postfixExp
      postfixExp = postfixExp + ch
      break
    case '(':      // Save '(' on the stack aStack
      aStack.push(ch)
      break
    case operator: // Process operators of higher precedence
      while (!aStack.isEmpty() and
             aStack.peek() is not a '(' and
             precedence(ch) <= precedence(aStack.peek())) {
             postfixExp = postfixExp + aStack.peek()
             aStack.pop()
      }
      aStack.push(ch)  // Save the operator
      break
    case ')': // Pop the stack until matching '('
      while (aStack.peek() is not a '(') {
        postfixExp = postfixEP + aStack.peek()
        aStack.pop()
      }
      aStack.pop()  // Remove the matching '('
      break
  }
}
```

```
// Append to postfixExp the operators remaining in the stack
while (!aStack.isEmpty()) {
  postfixExp = postfixExp + aStack.peek()
  aStack.pop()
}
```