–Sample–

CS 308 Data Structures

Spring 2003 - Dr. George Bebis

Final Exam

Duration: noon - 2:00 pm

Name:

1. **True/False** (3 pts each) **To get credit, you must give brief reasons for your answers !!**

(1.1) [✓ T] **F** An inorder traversal always processes the elements of a tree in the same order, regardless of the order in which the elements were inserted.

always in sorted order

(1.2) [✓ T] **F** The running time of Reheap-Down in $O(logN)$ where N is the number of elements in the heap.

In the worst case, Reheap-Down would have to traverse all the levels of the tree, making two comparisons per level. $\Rightarrow 2logN = O(logN)$

(1.3) **T** [✓ F] The largest value in a binary search tree is always stored at the root of the tree.

right-most node

(1.4) **T** [✓ F] An $O(logN)$ algorithm is slower than an $O(N)$ algorithm.

It is faster since logN increases slower than N

(1.5) [✓ T] F Inserting an element onto an unsorted list takes $O(1)$ time.

(for linked lists)

insert at the beginning of the list

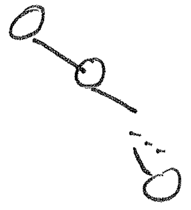
(or at the end for array-based lists)

(1.6) [✓ T] F To delete a dynamically allocated tree, the best traversal method is *postorder*

delete children first, then the parent.

(1.7) T [✓ F] When building a binary search tree, the order in which elements are inserted in the tree is unimportant.

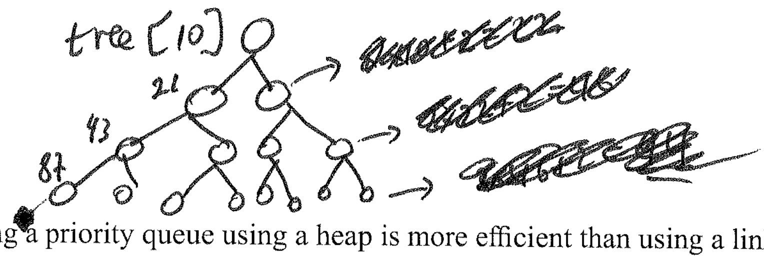
very important — ~~[illegible]~~ inserting in sorted order will yield a linked list!



(1.8) T [✓ F] Derived classes have access to the private members of their base classes.

only if ~~they~~ they have been declared ~~[illegible]~~ as "protected"!

(1.9) T [✓ F] Suppose that a complete binary tree containing 85 elements is stored in an array *tree[]*. The subtree rooted at tree[10] is a full binary tree with four levels.



(1.10) [✓ T] F Implementing a priority queue using a heap is more efficient than using a linked-list.

heap $\begin{cases} \text{Enqueue} \to O(\lg N) \\ \text{Dequeue} \to O(\lg N) \end{cases}$

list $\begin{cases} \text{Enqueue} \to O(N) \\ \text{Dequeue} \to O(1) \end{cases}$

Avg: $O(\lg N)$

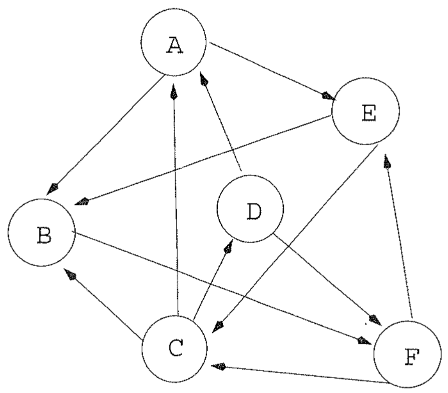
Avg: $O(N)$

2. **Short answers** (5 pts each)

(2.1) What is the height L of a full tree with N nodes ? Prove it.

$$N = \underbrace{2^0 + 2^1 + \cdots + 2^{L-1}}_{L \text{ levels}} = 2^L - 1 \Rightarrow 2^L = N+1 \Rightarrow \underline{\underline{L = \log_2 (N+1)}}$$

(2.2) Given the graph below, draw its linked-list representation (store the vertices in alphabetical order). Then, apply Breadth First Search (BFS) to find if there is a path from A to D (to get credit, you need to show all the steps clearly).



Same as in

Fall 2000 exam

(2.3) Explain the following terms:

activation record:

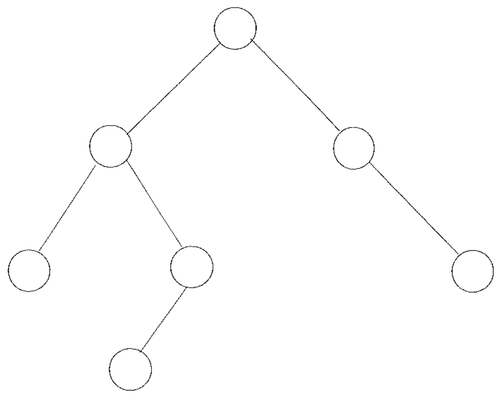
(see ~~[illegible]~~ notes)

run-time stack:

header and trailer nodes:

dynamic binding:

(2.4) Label the following binary tree with numbers from the set {6,22,9,14,13,1,8} so that it is a legal binary search tree (you can choose the numbers in this set in any order).



Similar to Fall 2000 exam...

Show how the tree above would look like after each of the following operations: (i) delete 13 (ii) insert 34 (use the original tree)

(2.5) Compare recursion with iteration in terms of (i) time, (ii) memory, and (iii) efficiency.

	Recursion	Iteration
Time	more time consuming (run-time stack)	less time consuming
Memory	more memory consuming (activation records)	less memory consuming
Efficiency	might solve same sub-problems many times	more efficient
	shorter code	longer code

(2.6) Trace the function below and describe what it does.

```
template<class ItemType>
int Mystery(TreeType<ItemType> *tree, int &n)
{
 if(tree != NULL) {
   n++;
   Mystery(tree->left, n);
   Mystery(tree->right, n);
 }
}
```

Same as in Fall 2000 exam

(4) (a) **[10 pts]** Define a new class *LookAheadStack* that is derived from class *StackType* (its definition is shown below). A look-ahead stack differs from the standard stack only in the push operation. An item is added to the stack by the push method only if it is different from the top stack element.

```
struct NodeType {
  int info;
  NodeType* next;
};

class StackType {
 public:
   StackType();
   ~StackType();
   void MakeEmpty();
   bool IsEmpty() const;
   bool IsFull() const;
   void Push(int);
   void Pop(int&);
 private:
   NodeType* topPtr;
};
```

See "Inheritance" slides

(we have done in class!!)

(b) **[10 pts]** Implement the new push function and the derived class's constructor.

(c) **[5 pts]** Which functions and from which class should be declared as *virtual*?

(5) **[15 pts]** Write a **client** boolean function that determines if a binary search tree ~~and an unsorted~~ sorted list contain exactly the same elements. Analyze its running time performance using big-O. The specifications of the binary search tree and unsorted linked-list are given in the next page.

```
bool MatchingItems(TreeType<ItemType> tree,
        UnsortedType<ItemType> list)
{ bool same= true;
  ItemType item;

  if (list.LengthIs() != tree.NumberOfNodes())      -> O(1)   -> O(N)
      same=false;                                    max
  else
    while (!list.IsEmpty() && same){                 -> N times
       list.GetNextItem(item);                       -> O(1)
    }  tree.RetrieveItem(item, same);                -> ~~O(N)~~ O(lgN)
                                                        (for balanced tree...)
  return same;
}
```

total: $N \times O(\lg N) = O(N\lg N)$

```
template<class ItemType>
class TreeType {
  public:
    TreeType();
    ~TreeType();
    TreeType(const TreeType<ItemType>&);
    void operator=(const TreeType<ItemType>&);
    void MakeEmpty();
    bool IsEmpty() const;
    bool IsFull() const;
    int NumberOfNodes() const;
    void RetrieveItem(ItemType&, bool& found);
    void InsertItem(ItemType);
    void DeleteItem(ItemType);
    void ResetTree(OrderType);
    void GetNextItem(ItemType&, OrderType, bool&);
    void PrintTree(ofstream&) const;
  private:
    TreeNode<ItemType>* root;
};

template<class ItemType>
class UnsortedType {
  public:
    UnsortedType();
    ~UnsortedType();
    void MakeEmpty();
    bool IsFull() const;
    int LengthIs() const;
    void RetrieveItem(ItemType&, bool&);
    void InsertItem(ItemType);
    void DeleteItem(ItemType);
    void ResetList();
    bool IsLastItem() const;
    void GetNextItem(ItemType&);
  private:
    int length;
    NodeType<ItemType>* listData;
    NodeType<ItemType>* currentPos;
};

template<class ItemType>
class NodeType {
  ItemType info;
  NodeType *next;
};
```