This exam is to be done without any materials other than one paper sheet of notes. There are 5 parts for a total of 104 points. Answer them all on the question sheets. Incomprehensible answers get zero points! Inefficient or poorly expressed code, and answers that do not display understanding of course content might not earn full or even any credit, even if logically right.

**Part 1 (26 points)—This page and the next**

1. (so proj/lab work will count) Fill in your ITSUNIX login name, one character per box.
   
   Ignore empty boxes.

2. (3 pts.) Below, draw an approximately balanced binary search tree whose nodes each contain one character from your login name. Use the alphabetic ordering of letters, numeric order for digits, and order every letter after (larger than) every digit. This ordering is the one gotten from numeric comparison of the ASCII char codes.

3. Write the smallest digit (0-9) that is NOT in your login name _____
   (3 pts.) Mark with ? those nodes that are accessed when the binary search algorithm is used to test if the digit you wrote is in the binary search tree you drew.

In a **binary search tree**, at every node \( n \), the following rules apply:

1. The entry in node \( n \) is greater than or equal to each entry in node \( n \)'s left subtree (so it may be equal to one of those entries).

2. The entry in node \( n \) is less than each entry in node \( n \)'s right subtree.

**Draw your binary search tree here:**

---

4 (4 pts.) Draw a heap-ordered binary tree that contains the same data. Reminder, for this exam: In a **heap-ordered tree**, at every node \( n \), the following rule applies: The entry in node \( n \) is greater than or equal to each entry in each of node \( n \)'s subtrees.
struct node
{
    unsigned char data; //unsigned so < and > apply to positive char codes.
    node *pleft; //address of the root of left subtree, NULL if none.
    node *pright; //address of the root of right subtree, NULL if none.
};

(4 pt. ea.) Code bodies of the 4 C/C++ functions specified by pre/post conditions below:

unsigned char smallest_in_search( const node * p )
//PRE: p!=NULL and p points to the root of a BINARY SEARCH TREE
//POST: The SMALLEST char in this binary search tree is returned.
{ //You write! (hint: look to your left)

}

unsigned char biggest_in_search( const node * p )
//PRE: p!=NULL and p points to the root of a BINARY SEARCH TREE
//POST: The BIGGEST char in this binary search tree is returned.
{ //You write!

}

unsigned char biggest_in_heap( const node * p )
//PRE: p!=NULL and p points to the root of a BINARY HEAP-ORDERED TREE
//POST: The BIGGEST char in this heap-ordered tree is returned.
{ //You write! (Please don’t laugh out loud.)

}

unsigned char smallest_in_heap( const node * p )
//PRE: p!=NULL and p points to the root of a BINARY HEAP-ORDERED TREE
//POST: The SMALLEST char in this heap-ordered tree is returned.
{ //You write!

}
Part 3 (22 points)

The following defines a class that implements a stack of floats using a dynamically allocated partially filled array.

```cpp
#include <stdlib> // Supplies size_t
class Part3Stack {
    public:
        Part3Stack(std::size_t init_size = 13) // Constructor.
            { data = new float[init_size];
                capacity = init_size;
                used = 0;
            }
        bool is_empty();
            // Pre: The object is properly constructed.
            // Post: true is returned if the stack is empty.
            // false is returned if the stack is non-empty.
        float pop();
            // Pre: The stack is non-empty.
            // Post: The top stack entry is removed and its value is returned.
            // Remark: An assertion is fired if the precondition is not met.
            // Remark: The pop( ) operation takes short, constant time.
        void push( float X );
            // Pre: The object is properly constructed.
            // Post: The given value X is pushed on the stack.
            // ImplementationRemark: If the array is full, expand()
            // is to be called to replace the array with one with
            // larger capacity.
            // Remark: The push( ) operation takes short, constant time except
            // when expand( ) is necessary.
    private:
        void expand();
            // Pre: The object is properly constructed.
            // Post: The original stack information as accessed by the
            // public member functions is unchanged, but the
            // capacity has been DOUBLED.
        float * data; // Invariant: data!=NULL.
        std::size_t used; // Invariant: The array prefix used is data[0..(used-1)].
        std::size_t capacity; // Invariant: The number of allocated elements=capacity.
};
```
1. (4 points) Write a complete and correct implementation of the member function Part3Stack::is_empty().

2. (6 points) Same for Part3Stack::pop()

3. (6 points) Same for Part3Stack::push(float X)

4. (6 points) Same for Part3Stack::expand()

#include "Part3Stack.h"
#include <cassert>
using namespace std;
Part 4 (12 points)

Project 5, the maze solver, made use of a linked list to store the squares that had been found so far during a search for a new solution. Write code to implement the function defined by the given pre and postconditions.

```c
struct node {
    int x;
    int y;
    node * next;
};
bool is_square_in_list( int tx, int ty, const node * pfirst )
// Pre: pfirst != NULL and it points to the first node of a linked list of nodes
//      This linked list represents a sequence of squares. Each square is
//      represented by its x and y coordinates stored in fields x, y of the node.
// Post: The function returns true if square with coordinates (tx,ty) is in the
//      sequence. Otherwise, the function returns false.
{ // (8 points) you write..
}
```

(4 points) What specifically prevents the maze solution finding algorithm from over and over again extending the path found so far into squares that are empty and in the maze, but are part of the path found so far? (Zero for an uninformative answer like “An if statement.”)
5. Consider this class, and write your answers to the right and/or below:
   class Sample {
   private:
       int MyDataMember;
   public:
       void AFunMember();
   };
   (1) What determines the lifetimes of instances of the data member MyDataMember of this class?
   (2) What code constitutes the scope of instances of the data member MyDataMember of this class. (I.E., where within the text of the C++ source program does “MyDataMember” signify this data member and not something else?)

6. Demonstrate the two-stacks algorithm (of Proj4) on: (1+((2-3)*4))
   Write your answer by crossing out popped values and redrawing both stacks each time a value would have been pushed over a crossed out entry. Make sure the final value is clearly shown.
7. A one-dimensional example of the sparse matrix (presented in the last 2 lectures) is a sparse array of floats: It consists of the same information as a length $N$ array of floats except only the non-zero entries are stored. Code suitable C/C++ struct/class data structure definitions, and explain in English the algorithm to retrieve the float value that would have been accessed by $A[I]$ if an ordinary array float $A[N]$; had been used instead.

8. Display the specific mergesort subproblems that the computer formulates and solves when it sorts the first 6 characters of your ITSUNIX user name, using the mergesort algorithm. Draw this display in the form of a tree: Draw one node for each activation of $\text{mergesort}()$ where that node shows the sequence of characters that the corresponding activation sorts. So each node will have up to two children: one for each recursive call that the corresponding activation makes. Underneath each unsorted sequence also write the sorted result that the corresponding activation returns to its caller.
9. Give two main reasons heap ordered balanced binary trees are useful to implement sorting efficiently (that result is called heapsort) and to implement priority queues efficiently. Your answer must say which nodes and how many of them are accessed when the computer does each data structure operation. (A balanced binary tree with \( n \) nodes will have height approximately \( C \times \log(n) \).)

10. Compare the use of a queue in Main and Savitch’s simulation of a car wash (where the cars line up in their order of arrival, like at a movie ticket office) with the use of a priority queue in discrete event simulation. Remember that in discrete event simulation, a future event is stored in the priority queue with priority equal to the negative of the time the event will occur. Hence earlier events are simulated before later ones.

The simulation software should, among other things, print, for each car, a line showing the time of day when the washing of that car was completed.

Answer the particular questions: In each of the two kinds of simulators, how does the software detect that it should simulate the completion of the washing of a particular car? Where does it get the simulated time of day so it can print the simulated time of day when the simulated washing of that particular car is completed?