## University of Nevada, Las Vegas Computer Science 477 Spring 2023

## Review 1

Do not look at the solutions until you have at least tried to work the problems.

1. A 2-dimensional array `int A[4][6]` is copied into a 1-dimensional array `B[24]` in row-major order. Assume that $A[0][0]$ is copied to `B[0]`. For any i and j, the value of `A[i][j]` will be copied to `B[x]` for some x. Express x as a function of i and j.

2. A compiler stores an array `A[10][5][20]` into main memory in column major order, with base address 2048, and each entry of A requires two places in main memory. Where in main memory is `A[i][j][k]` stored?

3. Consider the recursive subprogram:

```
void george(int n)
  {
   if(n > 1) george(sqrt(n));
  }
```

   Assume that the function sqrt(n) returns $\lfloor\sqrt{\texttt{n}}\rfloor$ What is the asymptotic time complexity of `george(n)`?

4. Fill in the code for the operators of a stack of integers implemented as an array.

```
const int N = // whatever

struct stack
 {
  int item[N];
  int size; // number of items on the stack
  // bottom of a stack is at item[0];
 };

void initialize(s&stack)
 {
 }

void push(s&stack,int i)
 {
 }

 bool empty{s&stack)
  {
  }
```

```
int pop(s&stack)
 {
 }
```

5. Fill in the code for the operators of a stack of integers implemented as a linked list.

```
struct stack
 {
  int item;
  stack*link;
 };

void initialize(stack*&s)
 {
 }

void push(stack*&s,int i)
 {
 }

bool empty{stack*s)
 {
 }

int pop(stack*&s)
 {
 }
```

6. Fill in the code for the operators of a queue of integers implemented as an array.

```
int const N = // whatever

struct queue
 {
  int item[N];
  int front; // item[front] is the front item
  int rear; // item[rear-1] is the rear item
 };

void initialize(queue&q)
 {
 }
```

```
bool empty(queue&q)
 {
 }

void enqueue(queue&q,int i)
 {
 }

int dequeue(queue&q)
 {
 }
```

7. The implementation given in 6. allows the possibility of *false overflow.* What is that?

8. Give four solutions to the false overflow problem.

9. For the given implementation of binary search tree of integers, fill in the operators insert and find.

```
struct treenode
 {
  int item;
  treenode*left;
  treenode*right;
 }

void insert(treenode*&bst,int i)
 {
  // allow duplicate items
 }

treenode*find(treenode*bst,int i)
 // return pointer to i if found, otherwise a null pointer
 {
 }
```

10. For these problems, use the implementation of binary search tree in the previous problem.

    (a) What does the following code do?

```
void mystery1(treenode*bst)
 {
  if(bst)
   {
    mystery1(bst->left);
    mystery1(bst->right);
    cout << bst->item << endl;
   }
 }
```

(b) What does the following code do?

```
bool mystery2(treenode*bst)
 {
  if(bst)
   return 1 + mystery2(bst->left)+mystery2(bst->right);
  else return 0;
 }
```

(c) What does the following code do? Assume that `maxint(int a,int b)` returns the maximum of its parameters.

```
bool mystery3(treenode*bst)
 {
  if(bst)
   return 1 + maxint(mystery3(bst->left),mystery3(bst->right));
  else return -1;
 }
```

(d) Write code for computing the sum of the items of bst.

```
int itemsum(treenode*bst)
 {
 }
```

## Solutions

1. Solution: the predecessors of `A[i][j]` are all `A[k][l]` for k in the range `0 ... i-1` and l in the range `0 ... 5`, as well as all `A[i][l]` for l in the range `0 ... j-1`. Then x is the total number of predecessors of `A[i][j]`, that is. `x = 6i + j`.

2. The predecessors of `A[i][j][k]` are `A[0][0][0] ... A[9][4][k-1],A[0][0][k] ... A[0][j-1][k]`, `A[0][j][k] ... A[i-1][j][k]`, a total of `50k + 10j + i` predecessors. Each value of A requires two spaces in main memory, so that offset is `100k + 20j + 2i`. The base address in main memory of `A[i][j][k]` is thus `2048 + 100k + 20j + 2i`.

3. Let $G(n)$ be the time complexity of `george(n)`. We have the recurrence $G(n) = G(\sqrt{n}) + K$, where $K$ is a constant. The solution is $G(n) = \Theta(\log \log n)$.

4. ```
   void initialize(s&stack)
    {
     s.size = 0;
    }

   void push(s&stack,int i)
    {
     assert(s.size < N);
     s.item[s.size] = i;
     s.size++;
    }

    bool empty{s&stack)
     {
      return s.size == 0;
     }

    int pop(s&stack)
     {
      assert(not empty(s));
      s.size--;
      return s.item[s.size];
     }
   ```

5. ```
   void initialize(s&stack)
    {
     s = nullptr;
    }

   void push(stack*&s,int i)
   ```

```
 {
  stack*temp = new stack;
  temp->item = newitem;
  temp->link = s;
  s = temp;
 }

bool empty{stack*s)
 {
  return s == nullptr;
 }

int pop(stack*&s)
 {
  assert(not empty(s)); // error if you try to pop an empty stack
  int sav = s->item;
  s = s->link;
  return sav;
 }
```

6.
```
 void initialize(queue&q)
 {
  q.front = 0;
  q.rear = 0;
 }

bool empty(queue&q)
 {
  return q.front == q.rear;
 }

void enqueue(queue&q,int i)
 {
  assert(q.rear < N}
  q.item[q.rear] = i;
  q.rear++;
 }

int dequeue(queue&q)
 {
  assert(not empty(q);
  int sav = q.item[q.front];
  q.front++;
  return sav;
 }
```

7. False overflow is when the rear item is at the rear of the array, but there is unused space at the front of the array.

8. (a) Make the array larger. (This can be done for some programming languages.)

   (b) Slide. Move the contents of the queue to the front part of the array.

   (c) Wrap. Make the array virtually circular.

   (d) Make N large enough that the rear item will never go beyond the rear end of the array. (This can only be done if we have, in advance, an upper bound on the number of times **enqueue** will be executed during the running of the program.)

9.
```
void insert(treenode*&bst,int i)
{
 // allow duplicate items
 if(bst == nullptr)
i {
   bst = new(treenode);
   bst->item = i;
  }
 else if(i < bst->item) insert(bst->left,i);
 else insert(bst->right,i);
}

treenode*find(treenode*bst,int i)
 // return pointer to i if found, otherwise a null pointer
 {
  if(bst == nullptr) return nullptr;
  else if(bst->item == i) return bst;
  else if(i < bst->item) return find(bst->left,i);
  else return find(bst->right,i);
 }
```

10. (a) Execution of `mystery1(treenode*bst)` causes the items of bst to be written in postorder.

    (b) Returns the number of items of bst.

    (c) Returns the height of bst.

    (d)
```
int itemsum(treenode*bst)
    {
     if(bst)
      return bst->item+itemsum(bst->left)+itemsum(bst->right);
     else
      return 0;
    }
```