Test Review – ELEC 278

Recursion

- Commonly used
- Must have terminating condition where you can break out of the recurse, otherwise will be recursing infinitely
- Typically, function calling itself until base condition is true to produce correct output

Structure

- Defined by struct keyword and a name
- Consist of fields which can be of any data type
- Structures can be declared with definition

```
struct time {
    unsigned char   hour;
    int             minute;
    int             second;
    char            am_or_pm;
} now, later;
```

Array – Fundamental Info Structure

- Correspond to chunk of memory
- Can be multidimensional (tables) follows syntax: array[row][column]
- Data stored in row major (all of row 0, then row 1, row 2…)
- Adding to beginning of table, all entries must be shuffled down by 1 spot
- Adding to middle, once location is known, shuffle remaining entries down

Linked Lists – Fundamental Info Structure

- Data structures composed of nodes and pointers
- Nodes will typically have a value and pointer to its next node and/or previous nodes
- Declare using struct, can use typedef command to improve concision
- Two types of linked lists: singly and doubly
    - Singly → points in one direction, traversal is a pain
    - Doubly → nodes point both forward and back, traversal is easier

```
// node structure
struct node
{
    struct node *nextnode;
    int nodval;
};
typedef struct node Node;
```

Figure 1: Structure of a node, has fields consisting of a pointer and a value which can be any data type. Doubly linked lists will have 2 pointers

Stacks

- Data structures that operate on a Last In First Out (LIFO) basis
    - Think a stack of cookies in a jar, take them out from the top
- Only the top of the stack needs to be tracked
- Can be implemented with either array or linked list (array preferrable if the fixed size is known)
- 2 main operations can be performed
    - Push, Pop
    - Push – Adding an element to the **TOP** of the stack
    - Pop – Removing an element from the **TOP** of the stack

Linked List Implementation

- Define node structure (Figure 1)
- Instantiate a head pointer, set equal to NULL
- Define **push** function with a data value argument *n*
    - Needs to know what to push

Steps (Figure 2)

```
Node *head = NULL;
void push (int d)
{
    Node *pn = (Node *)malloc(sizeof(Node));
    if (pn != NULL)
    {
        pn->data_val = d;
        pn->nextnode = head;
        head = pn;
    }// this does not return the value being pushed

}// push linked list
```

1. Create local node pointer *pn*, malloc memory for it
2. Check if the malloc was successful, if it was proceed
3. Set *pn* value to *n*, *pn* next to *head*
4. Set *head* to *pn*

Figure 2: Implementation of push operation with linked list for a stack.

- Define **pop** operations, can be void or a data type to capture value being popped
    - Doesn't need to know what is being popped, assume data value needs to be captured

    Steps (Figure 3)
    1. Check if the stack is empty (*head* == NULL), instantiate a variable to capture data
    2. If it is not empty, create a temporary pointer *ptemp*, set *head* to *ptemp* next node
    3. Set variable to *ptemp* value
    4. Free *ptemp*, return variable

```
int pop(void)
{
    int num;
    if (head != NULL)
    {
        Node *ptemp = head;
        head = ptemp->nextnode;
        num = ptemp->data_val;
        free(ptemp);
        return num;
    }
    else return 0;
}// pop linked list
```

Figure 3: Pop operation, linked list

Array Implementation

- does not need nodes, but an array with predetermined max size
- Initially need to set top of stack to -1
- Define **push** operation (needs a value *n*)

    Steps
    1. Check if top of stack is equal to max size
    2. If not, assign *n* to stack index top, increment top
- Define **pop** operation (needs a pointer input **value* to capture data)

    Steps
    1. Check if stack is empty (top == -1)
    2. If not, scan value to pointer **value*, decrement stack index

```
int stack[100];
int max = 99;
int top = -1;
```

```
int push (int n)
{
    if (top == max) return 0;
    stack[++top] = n;
    return n;
}// push array
```

Figure 4: Push operation using an array.

```
int pop (*value)
{
    if (top == -1) return 0;
    *value = stack[top--];
    return 1;
} // pop array
```

Figure 5: Pop operation, array

Queues

- Data structure that follows the First In First Out (FIFO) principle and items in between tail and head cannot be accessed
- Can be implemented with either linked lists or arrays
- Need to track the back and front of queue, possibly queue length
- 2 main functions to implement: insert and remove

Linked List Implementation

- Setup nodes as before
- Create pointer to front, back set both to NULL & counting variable
- Define **insert** operation, needs an argument

Steps
1. Create temporary node pointer *pnew*, check malloc worked
2. If it was, set *pnew* value to n, *pnew* next to NULL
3. Check if queue has anything (*end* != NULL), if true set *end* next to *pnew*
4. Check if queue is empty (*front* == NULL), if true set *front* to *pnew*, increase count

```
Node *front, *end = NULL;
int count = 0;
```

```
void insert(int n)
{
    Node *pnew = (Node *)malloc(sizeof(Node));
    if (pnew != NULL)
    {
        pnew->data_val = n;
        pnew->next = NULL;

        if (end != NULL) end->nextnode = pnew;
        if (front == NULL) front = pnew;
        count++;
    }
}
```

- Define **remove** operation, does not need argument, though may need one to capture data value
  Steps
  1. Using count, check if the queue is empty
  2. If not empty, create *ptemp* and set equal to *front*
  3. Save *front* value to variable, set *front* to *front* next
  4. Decrement count variable, free *ptemp*
  5. Check if *front* == NULL, if so *end* must also == NULL

Array Implementation

- Try to avoid, working with fixed length is difficult
- Must declare an array with a size, front and length variable
- Define **insert** operation
  Steps
  1. Determine if length of the array is the same as max size, if not continue
  2. Assign index (front+length) % max to n
  3. Increment length

```
void insert (int n)
{
    if (len == maxsize) return;
    Q[(front+len)%maxsize] = n;
    len++;
}// array insert
```

- Define **remove** operations, need a pointer *n* to capture value
  1. Check if the queue is empty (if length is 0)
  2. If not, copy value of the front to *n*
  3. Set front to front incremented modulo max
  4. Decrement length

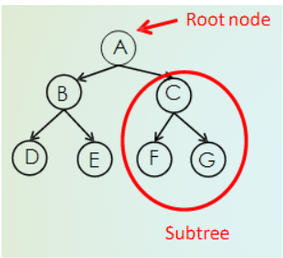
```
void remove (int *n)
{
    if (len == 0) return;
    *n = Q[front];
    front = (++front%maxsize);
    len--;
}// array remove
```

Deque

- Double ended queue, items can be added and removed at both ends
- Think line in a hallway, person in front and back could leave any time they wanted
- Must now implement 4 functions: add to front or back, remove from front or back
- Deque nodes must have a next and previous pointer

Trees

- Structure consisting of 0 or more nodes
- Implemented with nodes and pointers, this course focuses more on binary trees, though n-ary trees are possible
- Trees start out from a root node
- Can search through trees by looking for keys
- Binary trees occur when each node can have either 0, 1, or 2 children
  - No children – leaf
  - Children – left and right child



Terminology	Definition
Root	Beginning of tree, at level 0
Leaf	Tree node with no children
Binary Tree	Tree with nodes that may only have 2 children
Binary Search Tree	Binary Tree where left nodes are less than their root/ parent and right are greater
Key	Identifier for a node
Two Tree	BT that is either empty or every non-leaf has 2 subtrees that are two-tree

- Binary tree node structures will have 3 fields
  - A data value/ pointer to one
  - 2 tree node structures, one pointing to left, other pointing to right child

```
typedef struct node
{
    int value;
    struct node *leftChild, *rightChild;
} TreeNode;
```

- Need to function to **create tree nodes**, takes in a data value

  Steps
  1. Malloc size of node to a node pointer variable
  2. If successful, initialize node pointer fields (value, leftChild, rightChild)
  3. Return pointer

```
Node *makeTreeNode (int n)
{
    Node *pn = (Node *)malloc(sizeof(Node));
    if (pn != NULL)
    {
        pn->value = n;
        pn->leftChild = NULL;
        pn->rightChild = NULL;
    }
    return pn;
}
```

- For a binary tree populated with integers, the total sum of those integers can be found via recursion
- Create **function treetotal** that returns the sum of all integers in the tree
  - Start from the root node and work way down trees

    Steps
    1. Define a function that takes a treenode pointer as input **root*
    2. Initialize sum, left and right sum
    3. Check if the tree exists (*root* != NULL)
    4. Recursively calculate left and right sum by calling function with left and right child as input respectively
    5. Sum the 2 values + value of root node and return overall sum

```
int treetotal (Node *root)
{
    int leftSum, rightSum, sum;
    if (root == NULL) return 0;
    leftSum = treetotal(root->leftChild);
    rightSum = treetotal(root->rightChild);
    sum = root->value + leftSum + rightSum;
    return sum;
}// treetotal
```

- Create **function treeheight** to determine the height of a tree
  - Again, start from root, recursively determine height of left and right trees

    Steps
  - Height can be thought of the largest number of branches traversed to get to the furthest leaf
    1. Define function to take in a root node *root*
    2. Initialize a general height, left and right height variable
    3. Check that the tree exists (*root* is not NULL), if it is return height of empty tree (-1)
    4. Recursively call function to find height of left and right trees
    5. Compute height by adding 1 to the maximum of the left and right heights
    6. Return the height

```
int treeHeight (Node *root)
{
    int height, leftHeight, rightHeight;
    if (root == NULL) return -1;
    leftHeight = treeHeight(root->leftChild);
    rightHeight = treeHeight(root->rightChild);
    height = 1 + max(leftHeight,rightHeight);
    return height;
}// treeHeight
```

- Traversal, ordered method for visiting all nodes exactly once
  - Depth first and breadth
  - Depth can be subdivided into 3 categories
    1. Preorder – Root → Left → Right
    2. Postorder – Left → Right → Root
    3. Inorder – Left → Root → Right
  - Breadth – traverse by level, descending from the root
    - i.e. root → children → grandchildren → great grandchildren…
    - Can be accomplished using a queue

- Levels are an important concept, root has level 0
- Level also referred to as depth, height = max depth