Part A: The Basics

This section tests your basic knowledge of data structures via multiple choice questions. Sample questions include all the iclicker and reading quiz questions covered in class. Please make sure you review them.

Part B: Application, Comparison and Implementation

This section tests how well you understand what goes on under the hood of the data structures covered during the course, their strengths and weaknesses and their applications. The format is short answers and fill in the blanks.

1. B-trees.

(a) Construct a 2-3 tree by inserting the following keys in the order shown: 10, 15, 20, 25, 17, 30.
You can check your answers and experiment with trees of your own design at the following web sites:
https://www.cs.usfca.edu/~galles/visualization/BTree.html

(b) Which of the following are legal 2,3 trees (B tree of order 3)? For a tree that is not a valid 2,3 tree, state a reason why.

These two trees are valid 2,3 trees.

These two trees are not valid 2,3 trees.
In the left tree, the keys ”1” and ”7” are smaller than the root but this violates the key ordering requirement.
The right tree isn’t a valid 2,3 tree because there can be at most 2 keys in the root but there are 3.
2. Hashing.

(a) **Linear Probing.**
Using the hash function $H(i) = i \mod K$, a hash table of size $K = 7$ and linear probing, write in the state of the hash table after inserting the following elements: 13, 21, 11, 74, 33

(b) **Double Hashing.**
Perform the same insertion operations as in the previous question, but this time using double hashing instead of linear probing. $H_1(i) = i \mod K$, $H_2(i) = 1 + (i \mod (K - 1))$

(c) **Separate chaining**
Using the hash function $H(i) = i \mod K$, a hash table of size $K = 5$ and separate chaining. Write in the state of the hash table after insertion of the following elements: 13, 21, 11, 74, 33, 43, 21

(d) What is the load factor in part (c)?

\[ 7/5 \]

(e) Consider a hash table that uses separate chaining, and assume that the load factor is $N/M = \alpha$.

i. What is the average cost of a successful search?

\[ 1 + \alpha/2 \]

ii. What is the average cost of an unsuccessful search?

\[ \alpha \]
3. Entropy, Huffman trees and compression.

Let the probabilities of the five letters A, B, C, D, E be: \(\frac{1}{4}, \frac{1}{8}, \frac{1}{4}, \frac{1}{8}, \frac{1}{4}\). 

(a) Construct the Huffman tree for these five letters.

In order to ensure a unique tree, assign a letter to each node: the leaf nodes are assigned their associated letter, internal nodes are assigned the smaller letter of the two children (in alphabetical order). The left child of each node is the one associated with a smaller letter.

In order to ensure a unique encoding, the left child of each node corresponds to the zero bit and the right child corresponds to one.

(b) Encode the sequence ACEDCAA

```
<table>
<thead>
<tr>
<th>Letter</th>
<th>Code</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>00</td>
</tr>
<tr>
<td>B</td>
<td>111</td>
</tr>
<tr>
<td>C</td>
<td>10</td>
</tr>
<tr>
<td>D</td>
<td>110</td>
</tr>
<tr>
<td>E</td>
<td>01</td>
</tr>
</tbody>
</table>
```

001 001 110 100 000

(c) For the above string, compare the length of the encoded string with result obtained by by using 3 bits to encode each character (\(2^3 = 8\) is the smallest power of two larger or equal to 5). What is the achieved compression ratio?

\[
\frac{(3 \times 7)}{15} = \frac{7}{5}
\]
4. Consider the following set of initially unrelated elements is $S=\{0,1,2,3,4,5,6,7,8,9,10,11,12\}$. Assume that the initial union-find data structure is shown here, a forest of 12 singleton nodes with the header array initialized as shown.

\[
\begin{array}{cccccccccccc}
1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 & 10 & 11 & 12 \\
\hline
0 & -1 & -1 & -1 & -1 & -1 & -1 & -1 & -1 & -1 & -1 & -1 \\
\end{array}
\]

(a) Draw the final forest of up-trees that results from the following sequence of operations using union-by-size. Break ties by keeping the first argument as the root. Union(0,2), Union(3,4), Union(9,7), Union(9,3), Union(6,8), Union(6,0), Union(12,6), Union(1,11), Union(9,6)

Refer to review slides

(b) Draw the new forest of up-trees that results from doing a Find(4) with path compression on your forest of up-trees from (a).

Refer to review slides
5. Ternary search Trees.

(a) Consider the following ternary search tree. Nodes with double circles have their end bits set to true. Circle all of the words from the list on the right that are in the tree and write in any words that are missing. At the end you should have a complete list of all words found in the tree, and only those words.

```
air, all, a, and, as
```

(b) Does the height of this tree depend on the order in which the keys have been inserted?

Yes

(c) Briefly explain why you would prefer to use a ternary search tree rather than a binary search tree to implement `getAllValidWords()` in PA4.

The binary search tree doesn’t enable us to take advantage of a sequence of related search keys where each key is a prefix of all keys following it. As we search for a, an, and, we would have to begin each search fresh in BST. But in a ternary tree, as we progress along a path, we incrementally move from one key to the next. We don’t have to start the search anew for each key.

(a) Given the following AVL tree, insert the following keys into the tree, in the order shown: 3, 2, 6. We can’t award partial credit if any intermediate steps are missing. Circle the final tree so ensure that we know it is the outcome of the respective insertion operation.

(b) Which of the following trees have the AVL balance property? Circle all trees that meet the property. For any tree that you did not circle, explain why that tree isn’t an AVL tree.

1 and 2
Part C: Simulating Algorithms and Run Time Analysis

This section tests how well you understand the algorithms during the course. The focus is on simulating well known graph algorithms and the application of data structures to achieve fast run times. You will also be tested on analyzing the running time of these algorithms. The format is short answers and fill in the blanks.

1. **Graphs.** Given the following graph, run Dijkstra’s algorithm on it, with source node A. Alongside the graph is the data structure for each node that you will modify, see below.

   ![Graph Image]

   A: \( \text{dist} = 0 \quad \text{prev} = -1 \quad \text{done} = f \quad \text{adj: ((B,5),(D,10),(E,6))} \)

   B: \( \text{dist} = \infty \quad \text{prev} = -1 \quad \text{done} = f \quad \text{adj: ((D,3))} \)

   C: \( \text{dist} = \infty \quad \text{prev} = -1 \quad \text{done} = f \quad \text{adj: ((D,0))} \)

   D: \( \text{dist} = \infty \quad \text{prev} = -1 \quad \text{done} = f \quad \text{adj: ()} \)

   E: \( \text{dist} = \infty \quad \text{prev} = -1 \quad \text{done} = f \quad \text{adj: ((C,3))} \)

Note: While doing this problem, think about why the done flag is set after you dequeue a vertex from the priority queue and not when you enqueue it.

The boxes below represent the priority queue which is used in the algorithm. Each set of boxes represents the priority queue after the first pair has been dequeued and its associated node has been fully expanded (i.e. its neighbors, along with their distances, have been added to the priority queue as appropriate). You will fill these in.

Run Dijkstra’s algorithm by (1) modifying the values in the \textit{dist}, \textit{prev}, \textit{done} and \textit{done} fields (above) as the algorithm runs, and (2) showing the values stored in the priority queue after each new node is expanded.
For the data structures at the top, you should cross off old values as you replace them. For the priority queue, you should show the contents of the whole queue in the next set of boxes after you have fully expanded/explored from the node that you just removed from the front of the queue in the previous step. The initial priority queue and initial values are already filled in for you. You may stop the process when each node has been marked as done. You may not need all of the queues. If you do need more, just draw them in below the final array, to the side, or on the scratch page.

<table>
<thead>
<tr>
<th>(A,0)</th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
</table>

(initial priority queue)

<table>
<thead>
<tr>
<th>(B,5)</th>
<th>(D,10)</th>
<th>(E,6)</th>
</tr>
</thead>
</table>

(priority queue after exploring A)

<table>
<thead>
<tr>
<th>(D,8)</th>
<th>(D,10)</th>
<th>(E,6)</th>
</tr>
</thead>
</table>

<table>
<thead>
<tr>
<th>(C,9)</th>
<th>(D,8)</th>
<th>(D,10)</th>
</tr>
</thead>
</table>

<table>
<thead>
<tr>
<th>(D,10)</th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
</table>
2. For the graph given in part (a) construct an undirected version of the graph by replacing each directed edge by an undirected edge. Then construct the minimum spanning tree for the resultant graph. The MST will be the undirected version of the original graph with links of weight 10 and 6 removed.
3. Run Time Analysis

(a) Consider the following algorithm operating of a graph $G(V, E)$ and a source vertex $s$

i. Initially, set the visited field of all vertices in $V$ as FALSE

ii. Start at source vertex, $s$. Set the visited field of $s$ as TRUE, set counter to 1

iii. Enqueue $s$ into a queue

iv. While the queue is not empty and counter less than or equal to $|V|$  
   A. Dequeue the vertex $v$ from the head of the queue 
   B. For each of $v$’s neighbor vertex, $w$ (with visited FALSE):
      • Mark the visited field of $w$ as TRUE 
      • Increment counter by 1 
      • Enqueue $w$ into the queue

i. Identify the above graph algorithm.

BFS

ii. Assume the graph is implemented as an adjacency list. Write the Big O running time of each step of the algorithm (against each line of above) and derive the tightest overall Big O bound.

Refer to review slides
Part D: C++ and Programming Assignments

This section tests your understanding of C++ and the programming assignments Format: short answers and multiple choice.

1. Consider the classes below which represent a graph node (vertex) and a graph. Complete the code to perform breadth first search on the graph from the source node with index source. After BFS is called on the graph, all of the Vertex objects pointed to by nodes should have their members set as follows:

- visited should be true if this node is reachable from the source node, false otherwise
- prev should store the integer index of the node (vertex) from which this node (vertex) was first reached

Here are some helpful C++ commands and information of relevance here. You may not use all of these methods.

- You can access an element at position i in vector v using v[i]
- For an example of how to create and work with an iterator over a vector object, see the code provided to you at the start of BFS.
- Useful methods in class queue (Please do not use any other methods in the queue class. You do not need any others.):
  - q.front() accesses (returns) the next element in queue q (but not remove it)
  - q.pop() removes the next element (from the front) in queue q (but not return it)
  - q.push( d ) places the element d into the queue q (at the back).
  - q.empty() returns true if the queue q is empty, and false if there is at least one element in it

```cpp
1 class Vertex {  // Vertex node Class
2     public:
3         vector<int> adj;  // Indexes of vertexes directly connected to this vertex
4         bool visited;  // Has this vertex been visited?
5         int index;  // The vertexs index
6         int prev;  // The index of the vertex from which this vertex was found
7  }
8
9 class Graph {  // Graph Class
10     public:
11         vector<Vertex*> nodes;  // Pointers to vertexes in the graph
12         Graph( vector<Vertex*> theNodes ) : nodes(theNodes) {}
13  }
14
15 void BFS( int source ) {
16     vector<Vertex*>::iterator it = nodes.begin();  // initialize the graph
17     for ( ; it != nodes.end(); ++it ) {
18         (*it)->visited = false;
19         (*it)->prev = -1;
20     }
21     queue<Vertex*> toExplore;
```
toExplore.push(nodes[source]);
while (!toExplore.empty()) {
    Vertex* next = toExplore.front();
    toExplore.pop();
    vector<int>::iterator it = next->adj.begin();
    for (; it != next->adj.end(); ++it) {
        Vertex* neighbor = nodes[*it];
        if (!neighbor->visited) {
            neighbor->visited = true;
            neighbor->prev = next->index;
            toExplore.push(neighbor);
        }
    }
}