# CS261: Final Exam

**NAME:** ______________________________

**INSTRUCTIONS**

- This is a 90 minute, closed-book exam consisting of **FIVE** problems
- Cases of academic dishonesty will be filed to the Office of Student Conduct

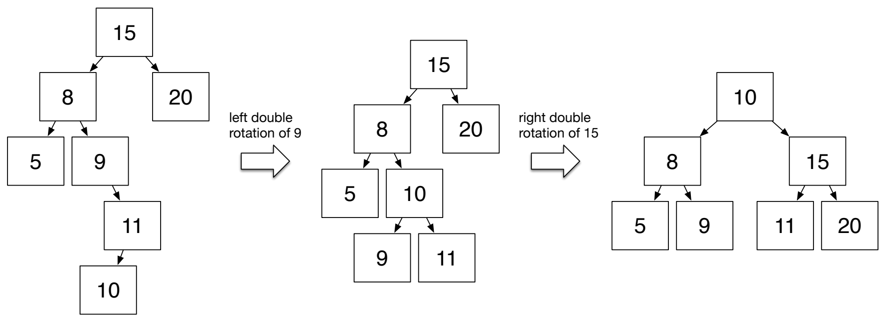
| Problems | Max points | Earned points |
|---|---|---|
| 1.1 | 10 | |
| 1.2 | 20 | |
| 2 | 20 | |
| 3 | 15 | |
| 4 | 15 | |
| 5 | 20 | |
| TOTAL | 100 | |

**Problem 1.1: AVL Trees – 10 points**

Transform the BST shown below to the corresponding AVL tree, and **draw** your final result.

Solution: The only way to do so is to height-balance all nodes bottom-up, starting from the leaves.

1) (0 points) A **height unbalanced** solution, or some nodes are missing from the BST.
2) (3 points) A drawing of a **height balanced** but incorrect AVL tree with **all** nodes from the BST.
3) (10 points) Start from 3 points, and add 1 point for every correct node in the solution that satisfies the table below.




| node | parent | left child | right child | points |
|---|---|---|---|---|
| 5 | 8 | NULL | NULL | 1 point |
| 8 | 10 | 5 | 9 | 1 point |
| 9 | 8 | NULL | NULL | 1 point |
| 10 | NULL | 8 | 15 | 1 point |
| 11 | 15 | NULL | NULL | 1 point |
| 15 | 10 | 11 | 20 | 1 point |
| 20 | 15 | NULL | NULL | 1 point |

### Problem 1.2: AVL Trees – 20 points

Write a C function, `rotateLeft()`, for the left rotation of a height-unbalanced node in an AVL tree. The nodes are specified by the code below. No other C functions are provided beyond this code.

```
# define TYPE int
struct Node {
  TYPE   val;
  struct Node *left;
  struct Node *right;
  int height;
};
```

```
/* return height of current*/
int height(struct Node *current){
  if (current == 0) return -1;
  return current->height;
}
```

```
/* Rotate left the input node
    - Input: current = height unbalanced node
    - Output: new height-balanced node
    - Pre-conditions: current and current->right are not NULL
*/
struct Node * rotateLeft(struct Node * current){
   /* FIX ME */

   struct Node * new = current->right;  /* 3 points */

   current->right = new->left;    /* 3 points */

   new->left = current;   /* 3 points */

   setHeight(current);  /* 2 points */

   setHeight(new);  /* 2 points */

   return new;   /* 2 points */
}

/* set height for current node */
void setHeight (struct Node * current) {
  int lch = height(current->left);  /* 1 point */
  int rch = height(current->right); /* 1 point */
  if (lch < rch)  /* 1 point */
    current->height = 1 + rch;  /* 1 point */
  else
    current->height = 1 + lch;  /* 1 point */
}
```

**Problem 2: Heap – 20 points**

Write a **recursive** C function, `containsHeap()`, that takes a heap and element `e` as input arguments, and returns -1 if `e` cannot be found in the heap; otherwise, returns the non-negative index of `e` in the heap. The heap is implemented as a dynamic array as specified below.

**Non-recursive solutions will be penalized by negative 10 points.**

```
#define TYPE int
#define EQ(a,b) (a == b)
#define LT(a,b) (a < b)
struct DynArr{
   TYPE *data;    /* pointer to the data array */
   int size;      /* number of elements in the array */
   int capacity;  /* capacity of the array */
};
/* Return the index of e in the heap, or return -1 if e is not in the heap */
int contiansHeap(struct DynArr *heap, TYPE e) {
   assert(heap);

   /* FIX ME */

   return _findIndexHeap(heap, e, 0);  /* 4 points */


}

/* Return the highest index of e in the heap, or  -1 */
int _findIndexHeap(struct DynArr *heap, TYPE e, int currentIdx){   /* 2 points */
  int leftChildIdx, rightChildIdx;

  if (currentIdx < heap->size) {        /* 2 points */
    if ( EQ(heap->data[currentIdx], e) )  /* 2 points */

      return currentIdx;    /* 2 points */

    else if ( LT(heap->data[currentIdx], e) ) {    /* 2 points */

      leftChildIdx = _findIndexHeap(heap, e, 2*currentIdx+1);    /* 2 points */
      rightChildIdx = _findIndexHeap(heap, e, 2*currentIdx+2);  /* 2 points */

      return (leftChildIdx > rightChildIdx ) ? leftChildIdx : rightChildIdx;  /* 2 points */
    }
  }
  return -1;
}
```

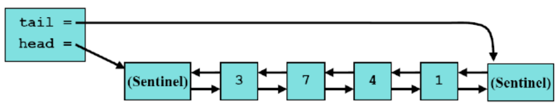
**Problem 3: 15 points**

Fill out the table with names of concepts or execution complexity of statements in the right column.

| | |
|---|---|
| $O(\log n)$ | Time complexity of finding an element in a Sorted Dynamic Array Bag with $n$ elements |
| Heap Sort | A sorting algorithm for a bag with $n$ elements that has complexity $O(n \log n)$ and uses an auxiliary data structure, but does not require extra memory space |
| Stack | An abstract data structure with the LIFO property |
| $O(n)$ | Time complexity of finding a given value in a Sorted Linked List with $n$ links |
| $O(n)$ | Time complexity of adding an element to a Sorted array with $n$ elements |
| If the right child exists: the leftmost descendant of the right child; otherwise: the left child | Which node do we copy to node $v$ in an AVL tree, when $v$ is to be removed |
| $O(\log n)$ | Time complexity of the remove-single operation for an AVL Tree with $n$ nodes |
| $O(1)$ | Time complexity of the add operation for a Hash Table with buckets implemented as a singly linked list, when the table size is $m$ |
| $O(1)$ | Time complexity of the remove-single operation for a Deque implemented as a doubly linked list with $n$ links |
| $O(1)$ | Time complexity of the remove-single operation for a Deque implemented as a wraparound dynamic array with $n$ elements |
| Leaves | Nodes of a complete tree that are guaranteed to respect the heap property |
| Dijkstra's algorithm | An algorithm for finding the minimum-cost path between two given nodes in a graph |
| Hash Table | An abstract data structure that is suitable for applications where the user rarely adds or removes data elements, but frequently searches for a given element |
| $O(2 * m) + O(n) = O(n)$ | Time complexity of the function that doubles the size of a Hash Table when the loading factor becomes too large, where the original table size is $m$ and the total number of elements in the Hash Table is $n$, $n > 2 * m$ |
| $O(n^3)$ | Time complexity of the Warshall's algorithm for a graph with $n$ nodes |

## Problem 4: 15 points

For the doubly linked list Deque defined and illustrated below, write the C function `_addDeque()` that adds a new element `e` **before** a given link `lnk` in the Deque, and as such can be used for adding *e* either to the head or tail of the Deque.




```
struct DLink  {
   TYPE val;
   struct DLink *next;
   struct DLink *prev;

};
struct Deque {
   struct DLink *head; /* Sentinel at front */
   struct DLink *tail; /* Sentinel at back */
   int size; /* Number of data elements */
};
```

```
/* Input: dq = pointer to a deque
          lnk = link in the deque before which the new element should be added
          e = the new element
   Post:  e is added before lnk,
          deque size is updated */

void _addDeque (struct Deque *dq, struct DLink *lnk, TYPE e){

   assert(dq && lnk);

   /* FIX ME */

   /* 5 points */
   struct Dlink * newlink = (struct Dlink *) malloc(sizeof(struct Dlink));
   assert(newlink != 0);


   newlink->val = e;              /* 1 point */
   newlink->prev = lnk->prev;     /* 2 points */
   newlink->next = lnk;           /* 2 points */
   lnk->prev->next = newlink;     /* 2 points */
   lnk->prev = newlink;           /* 2 points */
   dq->size++;                    /* 1 point */
}
```

**Problem 5: 20 points**

We are given the adjacency matrix $A$ of a directed graph with four nodes, as shown below. Every element $A[i][j]$ indicates a cost of the corresponding directed edge from node $i$ to node $j$ in the graph, where the infinity value $A[i][j] = \infty$ indicates that there is no edge from node $i$ to node $j$. For example, the cost of the directed edge from node 1 to node 3 is $A[1][3] = 2$, and there is no edge from node 3 to node 1.

Compute the min-cost directed path that starts at node 0 and ends at node 3. Write the total cost of your solution, and list the sequence of nodes along the min-cost path.

You may use any method by your choice for solving this problem, but you should explain your solution.

$$A = \begin{bmatrix} \infty & \infty & 1 & 10 \\ \infty & \infty & \infty & 2 \\ \infty & 1 & \infty & \infty \\ \infty & \infty & \infty & 10 \end{bmatrix}$$

Grading: Count the number of nodes along the path starting from 0 toward 3 (including 0 and 3) that are correct. Multiply that number with 4 points. The total minimum cost is additional 4 points.

Solution 1:

From $A$, there are only 2 paths between node 0 and node 3:

Path 1= $\{0 \to 3\}$, with the total cost = $A[0][3] = 10$

Path 2 = $\{0 \to 2 \to 1 \to 3\}$, with the total cost = $A[0][2] + A[2][1] + A[1][3] = 1 + 1 + 2 = 4$

It follows that the min-cost path is Path 2.

Solution 2: Dijkstra's algorithm:

Let $d[i]$ denote the cost of a min-cost path from node 0 to node $i$, and $prev[i]$ denote the previous node of $i$ along the min-cost path from node 0 to node $i$.

- Step1 : Initialize
  $d[2] = 1, \quad prev[2] = 0;$
- Step2 :
  $d[1] = d[2] + A[2][1] = 2,\ prev[1] = 2;$
  $d[3] = d[1] + A[1][3] = 4,\ prev[3] = 1;$

From $d[3]$, the minimum total cost is 4.

From $prev[]$, the min-cost path is: $3 \leftarrow 1 \leftarrow 2 \leftarrow 0$