EECE.3220: Data Structures

Spring 2017

Exam 3 Solution

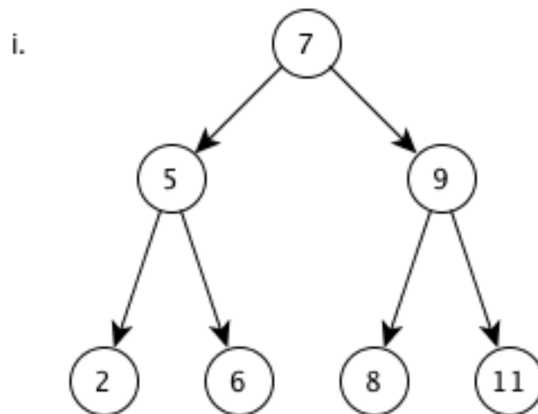
1. (28 points) ***Binary search trees***

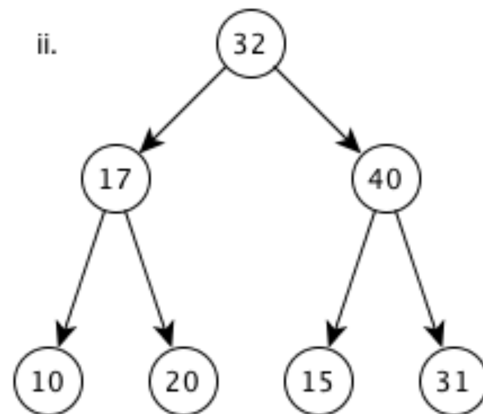
a. (6 points) Which of the trees below are binary search trees? **Circle ALL correct answers— there may be more than one.**

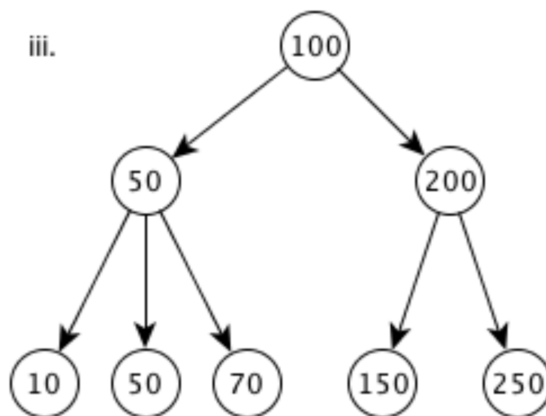
SOLUTION: *Choices (i) and (iv) are valid BSTs.*

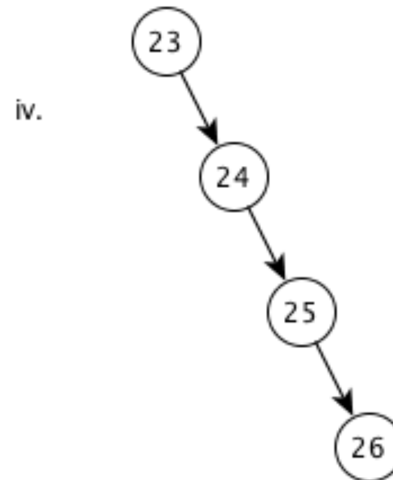
In choice (ii), node 15 violates the BST property, as it is less than the root of the tree (32) and therefore should be in the root's left subtree.

Choice (iii) is not a binary tree—node 50 has three children—and therefore cannot be a BST.









1 (continued)

b. (10 points) Given the program below, show (i) the contents of the BST this program generates (drawn like the trees shown in Question 1a), and (ii) the program output. Assume:

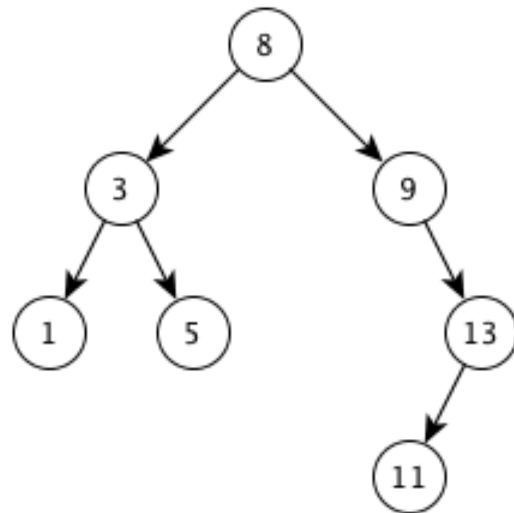
- This program uses the BST class definition on the extra sheet provided with the exam. That sheet also contains the definition of the `print()` function (and its helper function `printtree()`).
- The `add()` function adds a node to the given BST object as a leaf node in the appropriate location within the tree. You should not need a detailed `add()` definition to understand how nodes are added to a binary search tree.
-

```
int main() {
    BST test;

    test.add(8);
    test.add(3);
    test.add(5);
    test.add(1);
    test.add(9);
    test.add(13);
    test.add(11);

    cout << "Tree contents: ";
    test.print(cout);

    return 0;
}
```



SOLUTION:

(i) The BST created by the `add()` calls in this program is shown above.

(ii) The `print()`/`printtree()` functions perform a postorder traversal of the tree, visiting the left and right subtrees of each node, then finally printing the contents of the node itself. The program therefore generates the following output:

```
Tree contents: 1 5 3 11 13 9 8
```

1 (continued)

c. (12 points) The method for deleting a node from a binary search tree depends on the number of children that node has. The trickiest case is deleting a node, N, with 2 children, which requires the following steps:

   A. Identify the immediate successor of N and copy the contents of that node to N.
   B. Delete the immediate successor.

Complete the partial `delete()` function below. Your code should check if N has two children and, if so, identify its immediate successor and copy the contents of the successor to N. <u>You do not have to write any other part of the `delete()` function.</u>

<u>This problem uses the BST class definition shown on the extra sheet provided with the exam.</u>

```
void BST::delete(int v) {
   BNode *N;    // Pointer to node to be deleted
   BNode *S;    // Pointer to immediate successor of N

   ...      // Code to find node with value v in BST

   // SOLUTION STARTS BELOW THIS COMMENT

   // N points to node to delete; check if N has 2 children
   if (N->left && N->right) {  // Explicitly checking each ptr
                               //  != NULL is unnecessary but OK

      // Write code to set S = address of N's immediate successor
      S = N->right;         // Go right once,
      while (N->left)       //  then as far left as you can
         S = N->left;

      // Copy data from immediate successor to node N points to
      N->data = S->data;


   } // End of if statement

   // SOLUTION ENDS AT THIS COMMENT

   ...      // Code for remainder of delete function
}
```

2. (27 points) ***Heaps and priority queues***

a. (6 points) Which of the following statements about a priority queue are true? **Circle all true statements—there may be more than one correct answer.**

i. All valid priority queue implementations require that all items in the priority queue are sorted by priority.

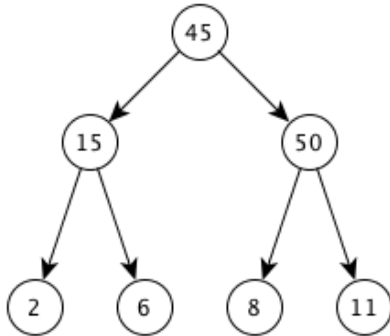
ii. If a priority queue is implemented using a heap, finding the highest priority item in the queue takes O(1) time.

iii. If a priority queue is implemented using a heap, removing the highest priority item in the queue takes O(1) time.

iv. Priority queues may be implemented using data structures other than a heap.

2 (continued)

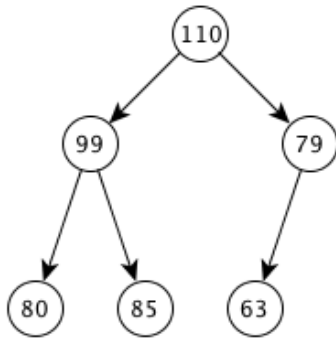
b. (9 points) For each of the binary trees below, indicate (i) whether it is a heap (by circling "Yes" or "No"), and (ii) if not, which heap property it violates.



(i) Is this tree a heap? Yes ***No***

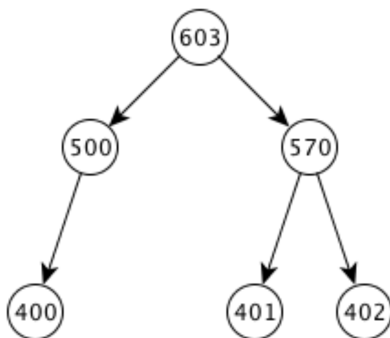
(ii) If no, which heap property does it violate?

Each node must be greater than its children (45 is less than 50, thus violating this property).



(i) Is this tree a heap? ***Yes*** No

(ii) If no, which heap property does it violate?



(i) Is this tree a heap? Yes ***No***

(ii) If no, which heap property does it violate?

Tree must be complete (500 has < 2 children; only second-level node that can have <2 children is rightmost node (570))

2 (continued)

c. (12 points) Inserting a new value into a heap involves two steps:

A. Place the new value at the end of the array.
B. "Percolate up" the new value by exchanging it with its parent until the new value is either (i) less than or equal to its parent, or (ii) moved to the root of the heap.

Use the space below to write the `percolate_up()` function as described above. The function takes a single argument, `pos`, that is the initial position of the value to be moved up the heap.

This function uses the Heap class definition shown on the extra sheet provided with the exam. You should assume that this Heap class is implemented similarly to the heaps discussed in lecture. Specifically:

- The first entry in the array that stores the actual data (`heaparr[0]`) is left empty and can be used as swap space if necessary.
- The root of the heap is stored in the second array location (`heaparr[1]`).

SOLUTION: *The code is shown below. The assumptions listed above make this function relatively simple to write because:*

- *Given a node at index `pos`, the parent of that node will always be at index `pos/2`.*
- *You'll keep swapping nodes until either (i) you've moved the value all the way to the root (`pos == 1`, making the 1st half of the loop condition false), or (ii) the node at position `pos` is less than or equal to its parent (making the 2nd half of the loop condition false).*

```
void Heap::percolate_up(unsigned pos) {
   while (pos > 1 && heaparr[pos] > heaparr[pos/2]) {

      // Swap the value at index pos with its parent
      heaparr[0] = heaparr[pos];
      heaparr[pos] = heaparr[pos/2];
      heaparr[pos/2] = heaparr[0];

      // Move up one level in the heap
      pos = pos/2;
   }
}
```

3. (20 points) ***Sorting***

Given the list of values below, show **all** of the steps required to sort the list using **either** quicksort or merge sort. **You do not have to show the results of both sorting algorithms.**

The following page is also blank to allow you extra space to show your work if necessary. The list of values to be sorted is reprinted on that page for your reference.

List of values to be sorted: {7, 9, 3, 1, 5, 10, 2, 12, 6, 4}

SOLUTION: *First, the steps for sorting with quicksort. This solution assumes the first value in the list or sublist is always the pivot:*

- *Sorting {7, 9, 3, 1, 5, 10, 2, 12, 6, 4} : Pivot = 7*
  - *Swap 4 & 9 → {7, 4, 3, 1, 5, 10, 2, 12, 6, 9}*
  - *Swap 10 & 6 → {7, 4, 3, 1, 5, 6, 2, 12, 10, 9}*
  - *Swap 7 & 2 →* ***{2, 4, 3, 1, 5, 6, 7, 12, 10, 9} → 7 in correct position***
  - *Now sort left sublist {2, 4, 3, 1, 5, 6} and right sublist {12, 10, 9}*
- *Sorting {2, 4, 3, 1, 5, 6} : Pivot = 2*
  - *Swap 1 & 4 → {2, 1, 3, 4, 5, 6}*
  - *Swap 1 & 2 →* ***{1, 2, 3, 4, 5, 6} → 2 in correct position***
    - ***Overall list at this point: {1, 2, 3, 4, 5, 6, 7, 12, 10, 9}***
  - *Left sublist {1} has only one value—no need to sort*
  - *Sort right sublist {3, 4, 5, 6}*
- *Sorting {3, 4, 5, 6}*
  - *Since this sublist is in order, I'll leave out steps to save space, but the quicksort function would recursively "sort" sublists {4, 5, 6} and {5, 6} before returning.*
- *Sorting {12, 10, 9} : Pivot = 12*
  - *Swap 12 & 9 →* ***{9, 10, 12} → 12 in correct position***
    - ***Overall list at this point: {1, 2, 3, 4, 5, 6, 7, 9, 10, 12}***
  - *No right sublist to sort*
  - *Sort left sublist {9, 10}*
- *Sorting {9, 10}*
  - *Quicksort function will recognize that sublist is in order, since (1) there are no values less than 9, the pivot, and (2) the right sublist {10} only has one value.*

The merge sort solution is on the next page.

3 (continued)

SOLUTION: *Now, the steps for using merge sort to sort {7, 9, 3, 1, 5, 10, 2, 12, 6, 4}. At each step, ordered sublists within larger lists will be underlined.*

- *Split {7, 9, 3, 1, 5, 10, 2, 12, 16, 4} into two lists by alternately writing ordered sublists into those two lists (i.e., {7, 9} goes into L1, {3} goes into L2, {1, 5, 10} goes into L1, and so on):*
    - *L1: {7, 9, 1, 5, 10, 4}*
    - *L2: {3, 2, 12, 16}*
- *Merge L1 and L2 together into a single list by merging sublists (one from each of L1 & L2) so the larger sublist they produce is also in order (i.e. {7, 9} and {3} are merged into {3, 7, 9}):*
    - *List: {3, 7, 9, 1, 2, 5, 10, 12, 16, 4}*
    - *# sublists = 3*
- *Split {3, 7, 9, 1, 2, 5, 10, 12, 16, 4} into two lists*
    - *L1: {3, 7, 9, 4}*
    - *L2: {1, 2, 5, 10, 12, 16}*
- *Merge L1 and L2 together:*
    - *List: {1, 2, 3, 5, 7, 9, 10, 12, 16, 4}*
    - *# sublists = 2*
- *Split {1, 2, 3, 5, 7, 9, 10, 12, 16, 4} into two lists*
    - *L1: {1, 2, 3, 5, 7, 9, 10, 12, 16}*
    - *L2: {4}*
- *Merge L1 and L2 together:*
    - *List: {1, 2, 3, 4, 5, 7, 9, 10, 12, 16}*
    - *# sublists = 1 →* ***SORT COMPLETE***

4. (25 points) ***Hash tables***

a. (5 points) You want to store data with the following key values in a hash table with 8 locations: {3, 6, 8, 12, 15, 16, 20}

If `k` refers to a key value and `L` refers to a location in the hash table (0 ≤ `L` ≤ 7), which of the following hash functions (some of which are written in pseudocode) causes the fewest collisions for this data set?

i. ***`L = k % 8;`***

ii.
```
s = 0.25 * k;
x = fractional part of s;      // i.e, if s = 5.25, x = 0.25
L = integer part of (x * 8);   // i.e., if x*8 = 1.75, L = 1
```

iii.
```
M = maximum key value;         // In this data set, M = 20
L = k / M;
```

iv. All three hash functions (i-iii) cause the same number of collisions for this data set

SOLUTION: *The table below shows the value of L (the location to which a given key value maps) for each key value and each hash function. For the purposes of this problem, we consider the number of collisions to be the number of keys that map to the same index as another key.*

	Hash functions		
Key values	L = k % 8	L = int(x * 8)	L = k / M
3	3	6	0
6	6	4	0
8	0	0	0
12	4	0	0
15	7	6	0
16	0	0	0
20	4	0	1
Collisions	4	6	6

4 (continued)

b. (10 points) Show the output of the program below, which uses the HashTable class definition and function definitions shown on the extra sheet provided with the exam.

```
int main() {
    HashTable <int>HT;

    HT.add(3);     HT.tab = {?, ?, ?, 3, ?, ?, ?, ?, ?, ?}
    HT.add(13);    HT.tab = {?, ?, ?, 3, 13, ?, ?, ?, ?, ?}
    HT.add(1);     HT.tab = {?, 1, ?, 3, 13, ?, ?, ?, ?, ?}
    HT.add(34);    HT.tab = {?, 1, ?, 3, 13, 34, ?, ?, ?, ?}
    HT.add(29);    HT.tab = {?, 1, ?, 3, 13, 34, ?, ?, ?, 29}
    HT.add(11);    HT.tab = {?, 1, 11, 3, 13, 34, ?, ?, ?, 29}
    HT.add(56);    HT.tab = {?, 1, 11, 3, 13, 34, 56, ?, ?, 29}
    HT.add(22);    HT.tab = {?, 1, 11, 3, 13, 34, 56, 22, ?, 29}

    HT.print(cout);

    return 0;
}
```

SOLUTION: *The code above has been annotated to show how each* `add()` *call changes the hash table, but it just shows the state of the* `HT.tab` *array. For each location in* `HT.tab` *in which a question mark is shown,* `HT.free` *will be true; otherwise,* `HT.free` *will be false.*

The `HT.print()` *function prints the contents of each location that is not free on its own line, so the output of this program is:*

```
1
11
3
13
34
56
22
29
```

4 (continued)

c. (10 points) Assume you have the same program as in Question 4b (reproduced below), but your HashTable class implementation (and the associated functions) has changed as follows:

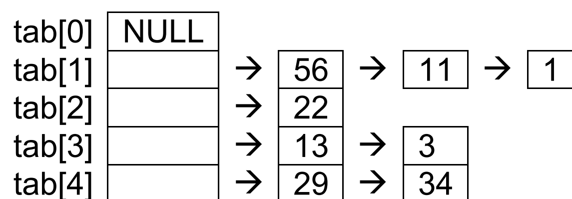
- The table uses chaining (placing multiple items that map to the same table index in a linked list) to resolve collisions
- Rather than the two arrays shown in the original `HashTable` class definition, the hash table with chaining uses a single array, `Node *tab[5];` where each entry in `tab[]` is a pointer to the head of a linked list. *(Note that this array only holds 5 pointers.)*
- Given a value, `v`, to be stored in the table, the hash function is simply `v % 5`.

How many values will be in the longest linked list in your hash table, and what will those values be? Show all work for full credit.

```
int main() {
    HashTable <int>HT;
    HT.add(3);          index = 3 % 5  = 3
    HT.add(13);         index = 13 % 5 = 3
    HT.add(1);          index = 1 % 5  = 1
    HT.add(34);         index = 34 % 5 = 4
    HT.add(29);         index = 29 % 5 = 4
    HT.add(11);         index = 11 % 5 = 1
    HT.add(56);         index = 56 % 5 = 1
    HT.add(22);         index = 22 % 5 = 2
    HT.print(cout);
    return 0;
}
```

SOLUTION: *The code above has been annotated to show what index each added value maps to. Values with the same index will be part of the same linked list. Therefore, the longest linked list (stored at index 1 in your table) will have **3 values**, and those values are **1, 11, and 56**.*

For those who prefer a graphical solution, one potential hash table is below. I say "potential" because the problem doesn't specify the order in which you place values in each linked list; this solution assumes each new value goes at the head of its linked list to save time.



EXTRA CREDIT

a. (5 points) In the space below, write the function `removeStar()`, which works as follows: given an input string, `s`, which contains a single `'*'` character, return a string with the `'*'` removed.

For example, `removeStar("*one") = "one"`, `removeStar("t*wo") = "two"`, and `removeStar("three*") = "three"`

SOLUTION: *Other solutions may be correct. The simplest way to write this function is to find the position of the asterisk, then concatenate the substrings before and after it.*

Recall there are two substring functions—`s.substr(p)` returns a substring within `s` starting at position `p` and going to the end of `s`, while `s.substr(p, n)` returns a substring within s starting at position `p` that is `n` characters long.

```
string removeStar(string s) {
   int pos = 0;

   // NOTE: pos = s.find('*') would do exactly what the loop
   //  below does, but we didn't cover that function in class

   while (s[pos] != '*' && pos < s.size() - 1)
        pos++;

   return s.substr(0,pos) + s.substr(pos+1);
}
```

EXTRA CREDIT

b. (5 points) In a game of war, the two players start with the decks shown below. Each card is listed as in Program 5, with a single character each for the rank and suit (rank is shown first).

Player A
`4c Th 8c 7c 8s Jd 5h Qc 2c 5d 6c 4h Kd`
`3s Td 2s 7h Ks Qd 9d Ah Ts Js Tc Jc 7d`

Player B
`4s Ac 5c 6s 8h 4d 3c Qh 2d 6h 3h 2h Kh`
`Ad Kc 3d 7s 9c Jh 5s As 9h 6d 8d Qs 9s`

Which player wins? For full credit, explain (briefly) how the game plays out.

SOLUTION: *Remember from Program 5 (or just from actually playing the game) that a "war" occurs when both players show cards of matching rank. In a war, each player deals three cards, then turns up one more card to be compared. If they're the same rank, then you have another round of war. If they're different, the player with the higher card wins everything on the table.*

The majority of this game plays out as a seven round war:

- A: **`4c`**, B: **`4s`** → WAR (A deals **`Th 8c 7c`**, B deals **`Ac 5c 6s`**)
- A: **`8s`**, B: **`8h`** → WAR (A deals **`Jd 5h Qc`**, B deals **`4d 3c Qh`**)
- A: **`2c`**, B: **`2d`** → WAR (A deals **`5d 6c 4h`**, B deals **`6h 3h 2h`**)
- A: **`Kd`**, B: **`Kh`** → WAR (A deals **`3s Td 2s`**, B deals **`Ad Kc 3d`**)
- A: **`7h`**, B: **`7s`** → WAR (A deals **`Ks Qd 9d`**, B deals **`9c Jh 5s`**)
- A: **`Ah`**, B: **`As`** → WAR (A deals **`Ts Js Tc`**, B deals **`9h 6d 8d`**)
- A: **`Jc`**, B: **`Qs`** → B wins the war!

Now, B has 51 cards, while A has just 1. On the next round, A turns up **`7d`**, *B turns up* **`9s`**, *B wins that round, and therefore **B wins the game.***