CS69001 Computing Laboratory – I

Assignment No: B2

Date: 08–August–2018

Let T be a binary tree with each node storing a key and two child pointers (left and right). Assume that the keys stored at the nodes are distinct from one another. Let n denote the number of nodes in T, and h the height of T. Your task is to convert the binary tree T to a binary search tree (BST) B such that:

(i) The keys in the BST B must be the same as the keys in the original binary tree T.

(ii) B must be structurally identical to T. Indeed, B should use the same memory allocated to store the nodes in T. The conversion of T to B must be *in place*, that is, no new memory is to be allocated. The conversion procedure would involve only swaps of keys among the existing nodes.

(iii) The conversion algorithm should run in $O(nh)$ time.

(iv) The conversion algorithm is allowed to use at most $O(h)$ extra space.

A simple algorithm that uses $O(n)$ extra space works as follows. Make an inorder traversal of T, and copy the keys in an array A. Sort A using any sorting algorithm. Make a second inorder traversal of T, and populate the keys of the tree in the sequence as they appear in the sorted array A. If you use an optimal algorithm to sort A (like merge sort or heap sort), the running time of this algorithm is $O(n \log n)$.

Your conversion algorithm is not allowed to use $O(n)$ extra space. The height h may be $\Theta(n)$ but may as well be as small as $\Theta(\log n)$. An external array storing the keys is thus not permitted. You should instead implement a variant of heap sort to solve your problem. Recall that a max-heap is a binary tree having two properties: (i) *heap structure* (the binary tree is complete), and (ii) *heap ordering* (each node stores a key no smaller than the keys stored in its two child nodes). In our case, T is not necessarily a complete binary tree. However, it makes sense to talk about max-heap ordering in T in the same sense mentioned above.

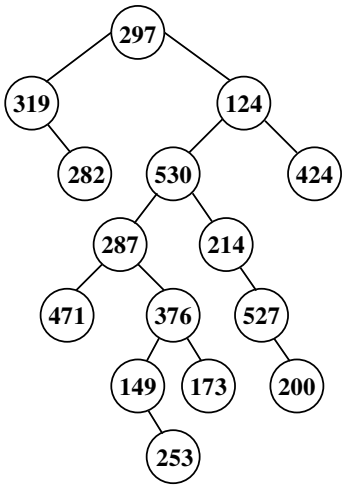
Standard **heap sort** uses a contiguous representation of the tree in an array H (this is natural in view of the heap structure property). The algorithm proceeds in two stages. First, H is converted to a heap. Second, a loop is executed in which the largest element (always at the beginning of the array) is deleted and is placed at the empty position at the end of the current heap, created by the deletion. A similar approach can be taken for an arbitrary binary tree T. Conversion of T to a heap (but only with the ordering property) is fairly straightforward. The second stage (the delete-store-and-heapify loop) is somewhat tricky. Nevertheless, since each heapify can be done in $O(h)$ time, the running-time bound is met. An extra $O(h)$ space is still needed to store the recursion stack.

Part 1: Declare a user-defined data type to store a node in your binary tree. Each node should contain only an integer-valued key and two child pointers (and nothing else).

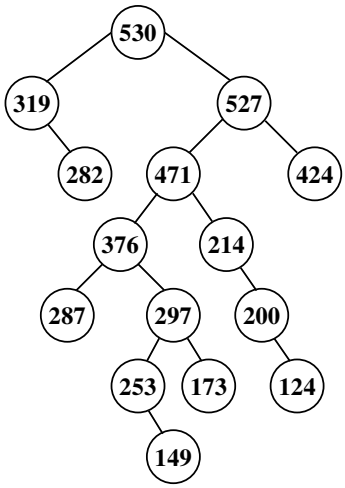
Write a function `constructtree` to build a binary tree T from user inputs. The user specifies each node by a triple (k, l, r), where k is an integer key to be stored in the node, and l and r are bits (1/0) indicating whether the node has a left child and a right child, or not. The user specifies the triples in a level-by-level and left-to-right (in each level) fashion. The input for the tree of Part (a) in the figure on the next page is illustrated in the sample output. You do not need the user to input the count n of nodes in T. Assume that it is the responsibility of the user to supply distinct keys for the different nodes.

A (FIFO) queue Q of node pointers is helpful in the construction process. Create a node for the root, and enqueue a pointer to this node to an initially empty Q. Subsequently, repeat the following steps until Q becomes empty. Locate the node at the front of Q, dequeue it from Q, read a triple (k, l, r) from user's input, populate the key field of the current node by k, create the requisite number of child nodes (depending upon the bits l and r), and enqueue the pointer(s) (if any) of the child node(s) to Q. Assume that the user supplies a syntactically and structurally correct set of triples, that leads to the construction of a unique binary tree T. Implement your own queue data structure. After this part, you never change any pointer in T.

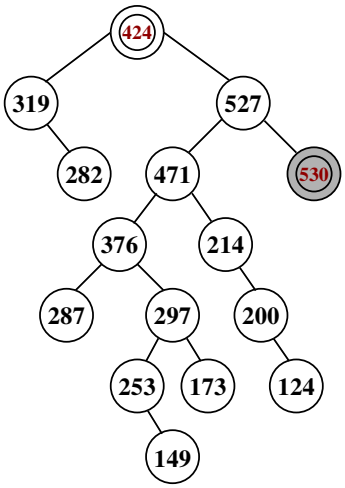
Part 2: Write two functions `preorder` and `inorder` to print the preorder and inorder listings of the keys of a binary tree. These two functions uniquely identify a binary tree, so these listings will help you debug.



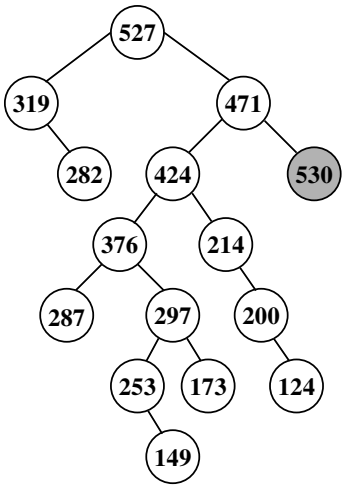
(a) The original binary tree



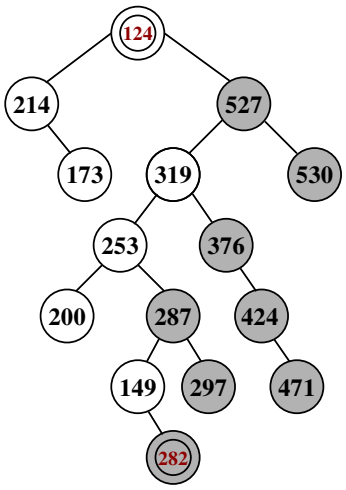
(b) Tree converted to a max−heap

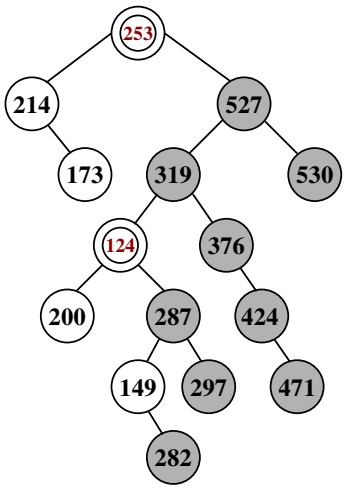


(c) Swap with root

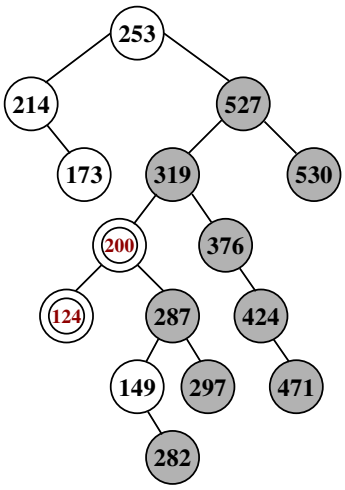


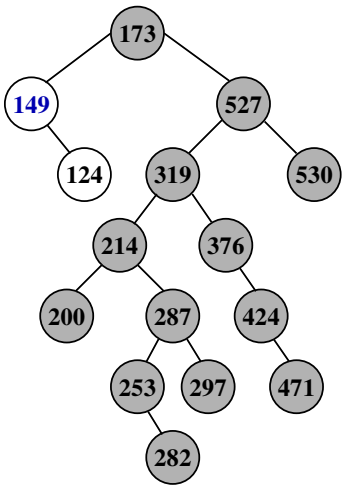
(d) Heapify at root





(e) Skip the "deleted" nodes





(f) Change of root

Part 3: Convert the binary tree to a max-heap with the heap ordering property (but having the same tree structure as created in Part 1). The process involves two functions **`heapify`** and **`buildheap`**. The function **`heapify`** takes a node v (a pointer to that, to be precise) as its only argument, assumes that the two subtrees of v are already heaps, and moves the key at v down the tree by swapping with child keys until the subtree rooted at v is converted to a heap. The other function **`buildheap`** works as follows.

```
if (v == NULL) return;
buildheap(v->left);
buildheap(v->right);
heapify(v);
```

The two recursive calls ensure that the two subtrees of v are already heaps. Subsequently, **`heapify`** moves the key of v to an appropriate position. Part (b) of the above figure shows the tree T converted to a heap.

Since each **`heapify`** runs in $O(h)$ time (h is the height of T), the entire heap can be built in $O(nh)$ time. However, if T is height-balanced, it is easy to see that **`buildheap`** actually takes $O(n)$ time.

Part 4: Now, write a function **`heaptoBST`** which is analogous to the second stage of heap sort. This essentially involves a loop, each iteration of which does the following four tasks.

(i) Identify the *last* position in the current heap.

(ii) Swap the key at this position with the key at the root (the root has the largest key in the current heap).

(iii) *Delete* the last position from the current heap.

(iv) Heapify at the root to restore heap ordering.

The *last* position of the current heap can be identified by a reverse inorder traversal of the tree. So your function should have the following structure. Here, T is the root and v is the current node in the traversal.

```
heaptoBST(T,v)
{
   if (v == NULL) return.

   heaptoBST(T,v->right).

   Here v is the "last" position, so Task(i) is solved.
   If v is the same as T
      Mark T as deleted (Task(iii)).
      Update the root T to T->left.
   else
      Swap T->key and v->key (this is Task(ii)).
      Mark v as deleted (Task(iii)).
      heapify at T (Task(iv)).

   heaptoBST(T,v->left);
}
```

Although conceptually simple, this procedure requires several points to consider.

***Marking a node as deleted*:** You are not allowed to store a "deleted" marker (internal or external) against each node, since that calls for $\Theta(n)$ extra space. This problem is solved as follows. After the keys of T and v are swapped (task (ii)), the node v contains the largest key l of the current heap. Any node storing a key $\geqslant l$ is to be considered deleted. So you need to keep track of the last relocated root key to identify the deleted nodes. In all subsequent heapify calls, nodes with *large* values are *ignored*.

The first iteration of the relocation-followed-by-restoration task is illustrated in Parts (b)–(d) of the given figure. Part (b) shows the initial heap. The reverse inorder traversal identifies the node storing 424 as the last position. It is swapped with the key 530 at the root (Part (c)). Now, the key 424 sitting at the root should be relocated because it violates heap ordering. Its two children have keys 319 and 527, so 424 should be swapped with 527. After this, 424 has two children 471 and 530. But 530 is deleted, so we swap 424 with 471, and heapify stops here (Part (d)).

***Skip the deleted nodes*:** You read it wrong, the deleted nodes should not be *ignored* but *skipped*. Look at Part (e) of the figure. The first tree in this part shows the keys after swapping 124 with 282. All nodes storing keys $\geqslant 282$ are virtually deleted, and shown as shaded nodes. Now, heapify at the root implies relocation of the key 124 at the root. In normal heapify, we look at the keys of the two child nodes. Here, these two keys are 214 and 527. But 527 is deleted (it is $\geqslant 282$). However, not all nodes in the subtree of 527 are deleted. Definitely, the reverse inorder traversal implies that all the nodes in the right subtree of 527 are deleted, but its left subtree may still contain undeleted nodes. So we keep on traversing left from 527. The first node encountered is 319 which is again deleted (along with its entire right subtree), so we move left further and find an undeleted node 253. So 124 should be swapped with the maximum of 214 and 253. In this case, the maximum is 253. The tree after this swap is shown in the middle tree of Part (e).

But 124 is too small to maintain heap ordering with its left child 200. This alone does not suffice. We should also keep on moving along left links in the right subtree, and try to meet the first undeleted node on the path. Here, it is 149 which is also larger than 124. But since $200 > 149$, we make a swap of 124 and 200, and heapify stops here (because 124 has reached a leaf node).

***The root may change*:** Finally, look at Part (f) of the figure. The reverse inorder traversal has reached $v = T$ (the recursive call on the right subtree returned). Now, there is no necessity to swap the keys of v and T, since they are the same node. In the subsequent recursive call on the left subtree, the root 173 will not change again. Instead, the left child 149 of the root is to be assigned the new root for this recursive call. Also, the earlier root 173 works as the new deleted limit marker.

It is clear from the above description that the function **`heapify`** written in Part 3 will not work for Part 4. Write a modified function **`heapifymodified`** to take care of the above observations.

The *main*() function

- Call **`constructtree`** to build the initial binary tree T by reading the level-by-level inputs of triples from the user.
- Print the preorder and inorder listings of the keys in T.
- Call **`buildheap`** to convert T to a (heap-ordered) max-heap. Print the preorder and inorder listings of the keys in the modified T.
- Call **`heaptoBST`** on T to generate a BST on T. Again, print the preorder and inorder listings of the keys in the BST so prepared.

Sample output

```
297 1 1
319 0 1
124 1 1
282 0 0
530 1 1
424 0 0
287 1 1
214 0 1
471 0 0
376 1 1
527 0 1
149 0 1
173 0 0
200 0 0
253 0 0

+++ Initial binary tree
    Preorder : 297 319 282 124 530 287 471 376 149 253 173 214 527 200 424
    Inorder  : 319 282 297 471 287 149 253 376 173 530 214 527 200 124 424

+++ After conversion to heap
    Preorder : 530 319 282 527 471 376 287 297 253 149 173 214 200 124 424
    Inorder  : 319 282 530 287 376 253 149 297 173 471 214 200 124 527 424

+++ After conversion to BST
    Preorder : 173 124 149 527 319 214 200 287 253 282 297 376 424 471 530
    Inorder  : 124 149 173 200 214 253 282 287 297 319 376 424 471 527 530
```

Submit a single C/C++ source file. Do not use global/static variables. Do not invoke STL features.