Algorithms and Data Structures
Week 5: Binary trees and binary search trees

Abstract

Summary: This week we'll finish off sorting and then start looking at a new data structure called trees. We'll see how to create and process data stored in a tree, and study how the shape of the tree has a big impact on the performance.

- **Book reference**: Drozdek 6.1–6.6.
- **Resources**: Week 5 archived eclipse program bundle.
- **Exercises:** Complete questions 1, 2, 3, 4 and 5.

  There is a submission box open in iLearn for these solutions. **Submission is entirely optional, but even if you do not submit it is strongly recommended that you attempt the questions.** Although there are no marks for submission, recall that the Week 6 in class test will be formed from a selection of the exercises you have been asked to study in these workshops.
- **Extension exercises:** Question 10. Submit this question in the individual Week 5 wiki.

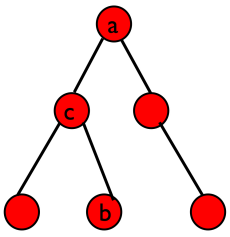
Trees

Trees can be used to represent parent/child relationships between data.

Trees consist of nodes connected by edges or arcs. The connections are directional, and there are no loops.

The ancestor of all the nodes in a tree is the root, and has no parents. A node without children is called a leaf.

Trees are hierarchical so that if c is between a and b, then c is the child of a and the parent of b. Also b is a descendent of a, and a is an ancestor of b.



Nodes may have several children. Trees such that all nodes have at most two children are called binary trees, and well be studying them for a while.

The definition of a tree is as follows. A tree T is a binary tree if:

1. T is empty, OR
2. T is not empty and is a subset of nodes such that
   (a) Exactly one node is the root;
   (b) All the other nodes are partitioned into two disjoint subsets of descendents called the left subtree, and the right subtree; each subtree is a binary tree.

Some useful definitions for analysing trees

The height of the tree

Let $H(T)$ be the height of a binary tree T, which we define as follows.

1. $H(T) = 0$, if T is empty;
2. $H(T) = 1 + \texttt{maximum}\{H(T_l), H(T_r)\}$, where T_l and T_r are the left and right subtrees of T.

Roughly speaking, the height of the tree is the number of levels when a tree is drawn neatly, with all nodes of the same generation on the same level.

Full trees

A binary tree is said to be full, if all nodes on level less than $H(T)$ have two children each.

Roughly speaking a full binary tree has no missing nodes.

Balanced trees

A binary tree is balanced, if the height of any nodes right subtree differs from the height of its left subtree by no more than 1.

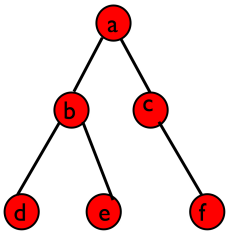
Tree traversals

Given a binary tree, we will be processing the items inside of it. To do it we will need to be able to "visit each item. There are three ways to traverse a binary tree (ie visit each item), and we call them preorder, postorder or inorder traversals.

preorder traversal: each node is "processed before the nodes in its subtrees;
postorder traversal: each node is "processed after the nodes in its subtrees;
inorder traversal: each node is processed in between the nodes of its left and right subtrees.
depth first traversal: The processing proceeds by visiting nodes, going as deeply as possible along paths. This is implemented using a stack of subtrees.
breadth first traversal: The procession proceeds level by level. This is implemented using a queue of subtrees.

Examples

Consider the following tree.



preorder output: a b d e c f
postorder output: d e b f c a
inorder output: d b e a c f
depth first output: a b d e c f
breadth first output: a b c d e f

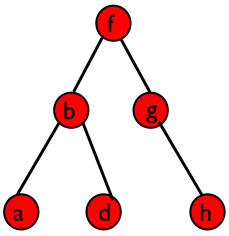
Binary search trees

Binary search trees, are a special type of binary tree in which searching is easy, because the nodes are all ordered relative to eachother.

A binary search tree (BST) is defined as follows. T is a binary search tree, if it is a binary tree, and the following BST conditions apply.

1. Ts root item is greater than all the node items of its left subtree, and
2. Ts root item is less than all the node items of its right subtree, and
3. both of Ts left and right subtrees are binary search trees (so that (1, 2) apply to their roots.)

Notice that an inorder traversal would print the items out in ascending order.



Searching *BST*'s

The basic idea is given a search key K and a tree T, we can recursively search the tree, just as in binary search, exploring either the left subtree or the right subtree according to whether K is less than depending on whether K is less than, or greater than the value at the current node.

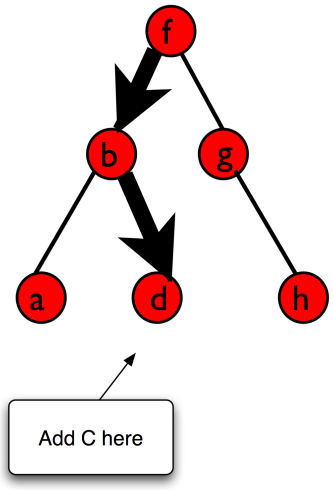
1. Start at the root of the *BST*
2. If K is equal to the root's data, then stop, otherwise
3. If K is less than the root's data then search the left subtree,
4. If K is greater than the root's data then search the right subtree.

The complexity of this algorithm depends on the shape of the tree. (Why?)

Creating *BST*'s (adding nodes)

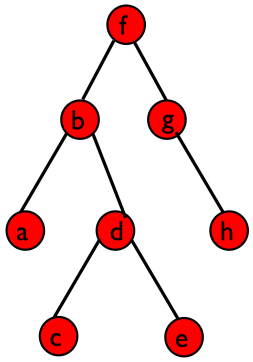
The basic idea is that well start from an empty tree and then just add items, making sure that the *BST* conditions are maintained. To do that suppose we already have a binary search tree T, and we want to add an item, x. We have to decide where in the tree to put it. (Well consider how to put it after weve figured out where to put it.)

An easy way to add a new item to a binary search tree is just to add it as a leaf node. To maintain the *BST* condition, we work our way down the tree — effectively doing a binary search with the new item as the key — ie taking the left branch whenever x is less than the current node, or the right branch when x is greater than the current node, until we reach a leaf. We then add it as a left/right child depending on whether x is less than or greater than the item at the leaf node.



Deleting nodes

Suppose we want to delete node 'b' from this tree. Obviously it will leave a gap, which must be filled by some other element already in the tree, but which one? 'd' would be a possibility, but then if we moved that node wholesale, we'd have to move 'a' somewhere else because, it can't become a third child of 'b'. We need to think again.



Case 1: Suppose we want to remove a leaf node, say 'h'. All we do is remove it — there's no more tidying up to do since the BST conditions in the remaining tree are not disturbed.

Case 2: Suppose we want to remove a node which does not have a left child, say 'g'. Now in this case it is fine to move the remaining subtree rooted at its right child to g's old position, because the BST conditions are not disturbed and no other node needs to be moved elsewhere.

The same idea works if there is no right child.

Case 3: That leaves the hard case, when a node has both a right and left child.

Suppose now we want to remove b. The idea is to take one of the other keys and copy it over b. But that doesn't solve everything for us, because we would still have to remove some other node. So we need to find a replacement key in the tree such that we still have a binary search tree and, after we have copied the node, the node we then have to remove can be solved by either Case (1) or (2) (because we know how to do it!) But which node should we choose?

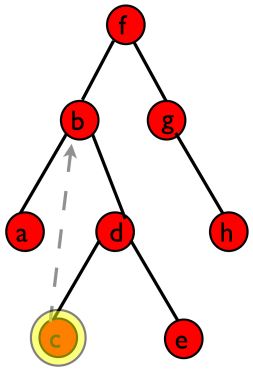
We choose one of two nodes.

Either we need a value which is greater than all the values in b's left subtree (in this case a), but less than all the values in its right subtree. (Why?) Such a value is found by taking the smallest value in (the former) 'b's right subtree. Recall that the smallest item in a BST is at the left-most node, in this case c. The diagram shows what happens when that node is a leaf. Once it has been copied then it can be removed just like Case 1. But it could have had a right child. In which case we would have had to remove it according to Case 2.

Or we could have chosen a value which is less than all the values in b's right subtree, but greater than all the values in the left subtree. This is found

analogously by taking largest value in the left subtree.

Either strategy will work.



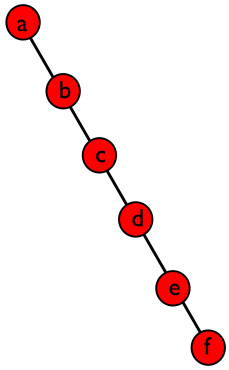
1. Search for the node to delete;
2. If it is not in the tree, do nothing;
3. If it is in the tree then decide whether Case 1, 2, 3 applies;
4. If Case 1, then remove the leaf;
5. If Case 2, then move the subtree up;
6. If Case 3, search for a node to copy which will not destroy the binary search tree structure; overwrite the node to be deleted with the appropriate node, then either delete the copied node (in case it is a leaf as in case 1), or move its subtree up (as in case 2). Why will Case 1 or 2 always apply?

Overall the deletion algorithm works by doing a binary search to locate the item to be removed, working out which of the three cases it belongs to, and then either deleting the node (in the case it is a leaf); OR replacing the node with its single subtree (in the case it only has one subtree);Or locating the smallest item in its right subtree as replacement item.

In all three cases the resulting tree is still a binary search tree.

Complexity analysis

The complexity of searching, insertion and deletion depends on the shape of the tree. In the diagram below, this tree gives the worst case complexity for all the examples that we have seen so far. (Why?)



In contrast a perfectly balanced tree gives the best performance. Since inserting and deleting nodes in a tree can spoil the balanced-ness of the tree sometimes it is best to "start again". The interesting thing is that all sets of nodes have an optimal insertion order to achieve a balanced tree; we can use this observation to design an easy method for rebalancing a binary search tree.

1. Copy all elements into an array;
2. Sort the items in the array;
3. Now create a new tree by recursively choosing the middle item to insert.

The Complexity of this is given by: the time to copy the items into the array plus the time to sort the items plus the time to re-insert all items into a new tree. This comes out to

$$O(N) + O(N \log N) + O(\log N^2) = O(N \log N) \ .$$

Note that if we chose to copy the items into a linked list, then an in order traversal will automatically process the list in a sorted form, allowing the list creation to be done efficiently (think of the Diagnostic). Using this observation can you think of a way to sort a linked list using a tree data structure?

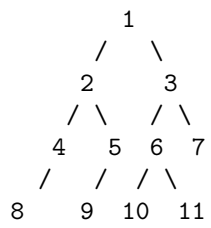
1 Workshop exercises

1. Trace the radix sort algorithm for the following list of letters.

   ```
   BCA, ABC, CAB, ACB, BAC, CBA
   ```

   Explain why it is important that the sorting algorithm for sorting on each individual "radix" must be stable for radix sort to sort the whole list correctly. Give an example to illustrate how a non-stable sort would not lead to a correct implementation of radix sort.

2. In a binary tree, a path between the root node and one of its descendents is the list of arcs connecting the two. If we write l for the left arc and r for the right arc then a path can be described as a string of l's and r. For example in the tree



the path connecting the root to f is rl, and the path connecting the root to i is lrl

Given a binary tree and a path, implement the function *followPath* which returns the value of the node which is connected to the root by the given path. Use the following specifications.

```
int followPath(IntBST t, String path);
// PRE: t is a binary tree; path is a string of l's and r's.
// POST: returns the value of the node connected to the root by path;
//       if there is no such node, returns -1.
```

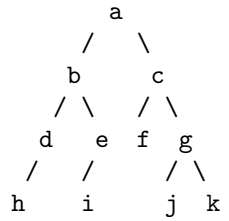
Make a binary tree class, based on the `IntBST` class demonstrated in lectures and implement and test your method.

3. In your binary tree class, write recursive functions for the following tasks.
   (a) Count the number of nodes in a binary tree;
   (b) Determine the maximum element;
   (c) Determine the sum of the elements (assuming the data items are integers);

4. Beginning with the empty binary search tree, what binary search tree is formed when you insert the following values in the order given? Use the insertion algorithm given in lectures.
   (a) W, T, N, J, E, B, A
   (b) W, T, N, A, B, E, J
   (c) A, B,W,J,N,T,E

   For each tree write down the sequence of letters obtained with an inorder, preorder and postorder traversal.

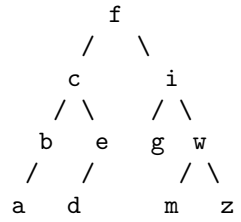
5. Consider the following tree



For each of inorder, preorder and postorder traversals, state the order of the nodes printed.

6. Now write a function which takes a binary search tree and adds the values of the nodes into a stack so that the items are sorted. You may use the standard functions of a basic stack class. (Does your implementation place the items so that they are sorted from bottom to top, or from top to bottom?)

7. Use the deletion function given in lectures to redraw the tree after deleting the following letters. First delete `i`, then `f`, then `a`. Draw the resulting tree after each deletion.



8. Arrange nodes that contain the letters `A, C, E, F, L, V` and `Z` into two binary search trees, one with maximum height and one that has minimum height.

9. Write a program that takes a binary search tree, and two items `a, b` with `a <= b` and prints out all the nodes have data lying between `a` and `b`.

10. The *dictionary* data type allows data to be accessed in various ways once it has been inserted into the dictionary. The following are some of the main operations that a dictionary should support:

    · *Search(D,k)* — Given a search key k, return the location (e.g. a pointer to) the element corresponding to k in dictionary D, if one exists.

    · *Insert(D,x)* — Given a data item x, insert it into the dictionary D, unless it is there already.

    · *Delete(D,p)* — Given the location of a data item p (e.g. a pointer to it) in the dictionary D, remove it.

Some dictionaries also support the following operations:

- *Max(D)* or *Min(D)* — returns respectively the items with the maximum or minimum key in D. This enables the dictionary to serve as a priority queue.
- *Predecessor(D,k)* or *Successor(D,k)* — Retrieve the item from D whose key is immediately before (or after) k in sorted order. These enable us to iterate through the elements in order.

(a) Design a dictionary data structure (not necessarily a tree) in which search, insertion, and deletion can all be processed in $O(1)$ time in the worst case. You may assume the set elements are integers drawn from a finite set $1, 2, ..., n$, and initialisation can take $O(n)$ time.

(b) Suppose now that there are n items stored in a balanced binary search tree data structure. Explain how search, minimum, and maximum can take $O(logn)$ time each. What do you need to do to this data structure to ensure that successor and predecessor take $O(1)$ time each.