CS3110 Spring 2017 Lecture 18: Binary Search Trees

Robert Constable

	Date for	Due Date
PS5	Out on April 10	April 24
PS6	Out on April 24	May 8 (day of last lecture)

1 Summarizing Computational Geometry

In this version of the course we have been able to demonstrate constructive synthetic geometric methods to find the convex hull of a collection of distinct points in the plane. The fist method we used was the "slow convex hull" algorithm, an n^3 method. This goal was explored in the spring 2016 version of CS3110 but was only partially achieved. On the other hand, as a class we explored this topic and deepened our understanding of it. We also found the little book by Donald Knuth called *Axioms and Hulls*[3]. Since that time, the PRL research group has been looking at this topic and seeking a constructive version of Knuth's work. Dr. Mark Bickford has just achieved this understanding by implementing Knuth's methods with the Nuprl proof assistant. He showed us in the previous lecture how to build the convex hull inductively, achieving an n^2 algorithm which is significantly better than the n^3 method used previously. We now see that in his small book Knuth has an $nlog(n)$ convex hull algorithm using Binary Search Trees. We have not yet been able to fully grasp this and make it constructive, if we can, we will have finally achieved the goal set in 2016. It will be a publishable result motivated and made possible by the interest of several students in two offerings of this course.

In the previous lecture, Ariel Kellison, a PRL staff researcher showed some of her results in using constructive synthetic methods to create a computational account of Euclidean geometry. We have provided pointers to some

of her work on this topic to help you see how Euclidean geometry can be constructively formalized. It is intriguing to think that perhaps some Greek geometer, say of the class of Archimedes, might have discovered results about convex hulls. We know that Archimedes discovered various geometric results that were previously thought to have been discovered only much later in developing the calculus.[1]

We now turn to discussing Binary Search Trees (BST's) for their own sake. There are some reasonable accounts of BST's on the web along with code for the algorithm. So it is not necessary to pack all of the important information into this lecture. We only discuss the highlights and repeat some of the implementation provided by Dr. Bickford using Nuprl.

2 Binary Search Trees (BST)

Binary search trees are a recursive data structure for storing elements of a set that allow for efficient operations to find, add, and delete elements. We require that we can decide equality on the keys used to identify the elements. In the next lecture we will store real numbers, for which we know equality is not decidable. So we store them as pairs of natural numbers and reals. We use the natural number as the key for looking up the real.

There is a good account of BST's in textbooks on algorithms, starting as early as Aho, Hopcroft, and Ullman *The Design and Analysis of Computer Algorithms* [1] and a more recent one in *Introduction to Algorithms* [2]. These textbooks present the algorithms in an imperative language using pointers rather than in a functional language using recursive data types.

The BST is a key data structure for storing and retrieving values efficiently. If n elements are randomly inserted into an empty BST we can expect the cost of operations to be O(n log n). Indeed, we can process a list of insert, delete and find operations on n distinct elements in O(n log n) time. The elements are organized in the tree so that at each node, the elements larger than the node element are in its right subtree and those smaller are in its left one. We show an example of this for 26 natural numbers in the range from 20 to 74. We see that at every node in the tree, all the numbers in the right subtree of this node have larger values and all those in the left subtree have smaller values. This property holds for each subtree.

[1] This discovery in relatively recent. It is known by the name of the book, Codex C – see wikipedia.org and look for Archimedes Palimpsest.

2.1 basic operations

To find whether an element k is in a given bst, first check k against the root. If k equals the root we know. If k is larger, then we ignore the left subtree and look in the right one. If k is less than the root, we look in the left subtree.

We add elements by a similar method. To add an element k to an empty bst, we create the root with this value. If the bst is non-empty, we compare k to the root value, r. If k > r, we add k to the right subtree, otherwise to the left one. It is possible depending on the "order of arrival" of an element that this process will simply create an ordered list of numbers, ascending or descending. In our example below, if the values are presented in the order 74,70,69,68,67,66,65,64,63,62,61,60,59,58,57,55,52,51,50,32,30,29,26,24,23,20 then the BST degenerates to the above list of elements in decreasing order, a degenerate tree.

The most delicate task is *deletion* of an element from a tree. We need to make sure that deleting an element does not destroy the order invariant described above, that is, at every subtree, all the elements in the left subtree are smaller than the root and all those in the right are larger than the root. There are two ways to delete an element. Deleting a leaf is easy, we just remove it. In our running example in the diagram, the leaf deleting applies to nodes, left to right, with the values 20, 24, 29, 32, 51, 55, 59, 62, 64, 66, 69, and 74.

To delete a node with exactly *one child*, such as 61, we remove it and replace it with 62. Likewise for node 68, replace it with 69. The interesting case is deleting a node with two subtrees, both left and right. There are two correct deletion methods.

1. Replace the deleted element with the *maximum element in its left subtree*, call it **max-left**. So to delete 57, we get a node with value 55, a right subtree with root value 58, and a left subtree with value 52.

2. Replace the deleted element with the *minimum element of the right subtree*, call it **min-right**.

Using this to delete 57, we replace it by 58 and have 58 point to 60. It is easy to see that these two methods of deletion work. We only look at the max-left case. Let's call the element we are deleting d. All the elements in the left subtree are smaller than the subtree root d we are deleting. The largest among these is the maximum of the left subtree, call it ml for "maximum

left". We also know that ml is smaller than all of the elements in the right subtree since they are all larger than d. So if we replace d by ml, then the right subtree invariant continues to hold. Moreover, since ml is larger than all of the other elements of the left subtree, when we replace d by ml, this remains the case. So the tree invariants for the left and right subtrees of d hold.

It would be a good exercise to prove that min-right also works. In preparing for the next lecture, we want to look at the OCaml definition of a binary search search tree.

2.2 recursive definition of bst and related notions

Recall the OCaml recursive definition of a polymorphic list from Lecture 4.

'a list = Nil — Cons of ('a * 'a list)

We use the same pattern for an 'a bst.

'a bst = Null — Leaf of 'a — Node of ('a bst * 'a * 'a bst)

In another lecture we will discuss *lazy versions* of these types which lead to the notion of a *stream* as a generalization of a list. Here is a stream type.

'a stream = 'a * (a' stream)

Here is a *spread* or lazy tree.

'a spread = ('a spread * 'a * 'a spread)

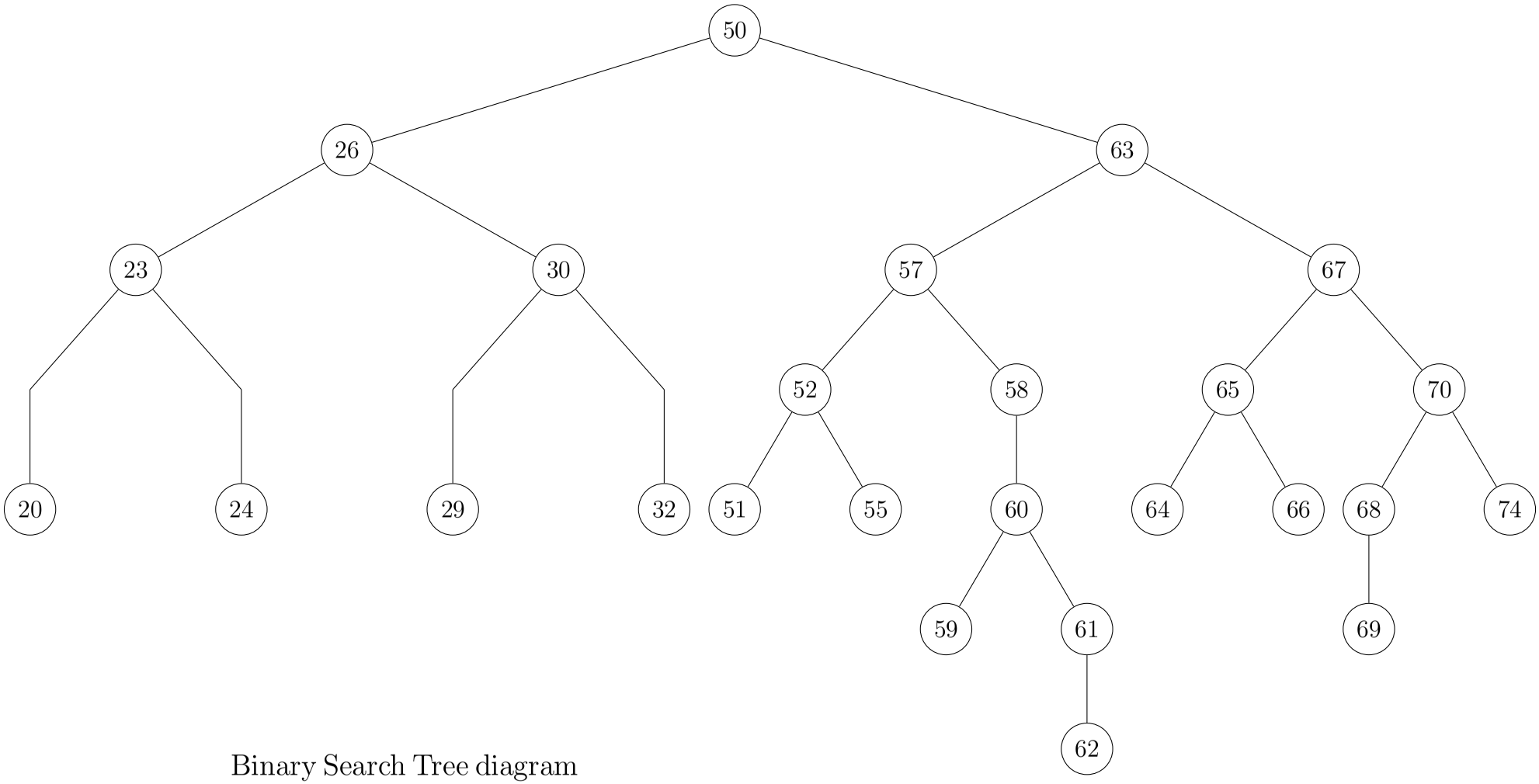
These are fascinating types that are used in the proof assistants and in some programming languages such as Haskell. We will study them even though they are not supported in the OCaml we are using.

References

[1] A. V. Aho, J. E. Hopcroft, and J. D. Ullman. *The Design and Analysis of Computer Algorithms*. Addison-Wesley, Reading, MA, 1974.

[2] Thomas H. Cormen, Charles E. Leiserson, and Ronald L. Rivest. *Introduction to Algorithms*. MIT Press/McGraw-Hill Book Company, Cambridge, Massachusetts, 1994.

[3] Donald E. Knuth. *Axioms and Hulls*. Lecture Notes in Computer Science 606, 1992.



Binary Search Tree diagram