CS1200: Intro. to Algorithms and their Limitations Prof. Anurag Anshu

Lecture 6: The RAM Model

Harvard SEAS - Spring 2026 *2026-02-10*

1 Announcements

- SRE2 today.
- Anurag OH after class.
- No class next Monday (President's day)

2 Recommended Reading

- Hesterberg–Vadhan, Chapter 5
- CLRS Sec 2.2

3 Loose end: Insertion in a BST

We concluded the last lecture with the following table:

Data structure runtimes		
Queries	Sorted Array	Binary Search Tree
`search`	$O(\log n)$	$O(h)$
`predecessor`	$O(\log n)$	$O(h)$
`successor`	$O(\log n)$	$O(h)$
`min`	$O(1)$	$O(h)$
`max`	$O(1)$	$O(h)$
`rank`	$O(\log n)$	$O(n)$ [$O(h)$ using augmentation]
`insert`	$O(n)$	$O(h)$
`delete`	$O(n)$	$O(h)$

Table 1: List of query runtimes for sorted arrays and binary search trees.

Theorem 3.1. *Given a binary search tree of height h, all of the* DYNAMIC PREDECESSORS & SUCCESSORS *operations (queries and updates), as well as* `min` *and* `max` *queries, can be performed in time $O(h)$.*

Proof. • **Insertions:**

```
Insert($T$,($K$, $V$)):
if $T = \emptyset$ then

else
    Let $v$ be the root of $T$, $T_L$ its left-subtree, and $T_R$ its right-subtree;
    if $K \leq K_v$ then



    else



```

Algorithm 3.1: `Insert()`

Runtime:

Proof of Correctness:

- **Search:**

  The algorithm is as follows:

- **Minimum (and Maximum):**
  The algorithm is to move to the left child until we reach a vertex with no left child, and then output the key at that vertex.

- **Predecessor (and Successor):**
  The algorithm for next-smaller follows along related lines to search algorithm:

- **Deletions:** The algorithm for deletion is the subject of today's Sender–Receiver Exercise.

□

4 Loose end: Balanced Binary Search Trees

So far we have seen that a variety of operations can be performed on binary search trees in time $O(h)$, where h is the height of the tree. Thus, we are now motivated to figure out how we can ensure that our binary search trees remain shallow (e.g. of height $O(\log n)$ where n is the number of items currently in the dataset). While doing so, we need to be sure that we retain the BST property.

There are several different approaches for retaining balance in binary search trees. We'll focus on the data structure produced by one such approach that is relatively easy to describe, called AVL trees, after mathematicians named Adelson-Velsky and Landis.

Definition 4.1 (AVL Trees)**.** An *AVL Tree* is a binary search tree that is:

1.

2.

Lemma 4.2. *Every nonempty height-balanced tree with n vertices has height at most $2 \log_2 n$.*

5 Computational Models

So far, our conception of an algorithm has been informal: "a well-defined procedure for transforming inputs to outputs" whose runtime is measured as the number of "basic operations" performed on a given input. This is unsatisfactory: how can we identify the fastest algorithm to solve a given problem if we don't have agreement on what counts as an algorithm or as a basic operation?

To address this, we need to specify a *computational model*. A computational model is any precise way of describing computations. In approximate order from "far from physical" to "close to physical," some examples of computational models are:

0. Natural language (e.g. 'calculate the prime factorization of the following number'):
1. Declarative and functional programming languages:
2. Imperative high-level programming languages:
3. "Close to the metal" imperative programming languages:
4. Architecture-level models:
5. Hardware models:

Translation from higher-level to lower-level models:
Also need a complexity measure to model "time" (or other resources).
What do we want from a computational model and a complexity measure?

- Unambiguity.
- Expressivity.
- Mathematical simplicity.
- Robustness.
- Technological relevance.

6 The RAM Model

Our first attempt at a precise model of computation is the *RAM model*, which models memory as an infinite array M of *natural numbers*.

Definition 6.1 (RAM Programs: syntax)**.** A *RAM Program* $P = (V, C_0, \ldots, C_{\ell-1})$ consists of a finite set V of *variables* (or *registers*), and a sequence $C_0, C_1, \ldots, C_{\ell-1}$ of *commands* (or *lines of code*), each chosen from the following:

- (assignment to a constant)
- (arithmetic)
- (read from memory)
- (write to memory)

- (conditional goto)

In addition, we require that every RAM Program has three special variables:

Definition 6.2 (Computation of a RAM Program: semantics)**.** A RAM Program $P = (V, (C_0, \ldots, C_{\ell-1}))$ *computes* on an input x as follows:

1. Initialization:

2. Execution:

3. Output:

The *running time* of P on input x, denoted $T_P(x)$, is defined to be:

The definition of the RAM Model above is mathematically precise, so it achieves our unambiguity desideratum (unless we've forgotten to specify something!).

The RAM Model also does quite well on the mathematical simplicity front. We described it in one page of text in this book, compared to 100+ pages for most modern programming languages. That said, there are even simpler models of computation, such as Turing Machines and the Lambda Calculus. However, those are harder to describe algorithms in and less accurately describe computing technology. We will briefly discuss those later in the course when we cover the Church–Turing Thesis, and they (along with other models of computation) are studied in depth in CS1210.

Our focus for the rest of this chapter will be to get convinced of the *expressivity* of the RAM model. We will do this by seeing how to implement algorithms we have seen in the RAM model. We will turn to robustness and technological relevance next time.

7 Iterative Algorithms

Theorem 7.1. *Algorithm 7.1 solves* Sorting On Natural Numbers *in time $O(n^2)$ in the RAM Model.*

Our goal in the rest of this lecture is to get convinced that all of the algorithm and data structure analyses we have done so far in the course can analogously be made completely precise and rigorous in the RAM Model. We present the low-level RAM code to convince you that this can be done in principle, but outside that context one does not generally read or write low-level RAM code (because that would be tedious).

```
InsertionSort($x$):
Input      : An array $x = (K_0, V_0, K_1, V_1, \ldots, K_{n-1}, V_{n-1})$, occupying memory
             locations $M[0], \ldots, M[2n-1]$
Output     : A valid sorting of $x$ in the same memory locations as the input
Variables  : input_len, output_len, zero, one, two, output_ptr, outer_key_ptr,
             outer_rem, outer_key, inner_key_ptr, inner_rem, inner_key, key_diff,
             insert_key, insert_value, temp_ptr, temp_key, temp_value
0  zero = 0 ;                                                /* useful constants */
1  one = 1;
2  two = 2;
3  output_ptr = 0 ;                                   /* output will overwrite input */
4  output_len = input_len + zero;
5  outer_key_ptr = 0 ;                          /* pointer to the key we want to insert */
6  outer_rem = input_len/two ;                  /* # outer-loop iterations remaining */
7      outer_key_ptr = outer_key_ptr + two ;                   /* start of outer loop */
8      outer_rem = outer_rem − one;
9      IF outer_rem == 0 GOTO 34;
10     outer_key = $M$[outer_key_ptr] ;                         /* key to be inserted */
11     inner_key_ptr = 0 ;                     /* pointer to potential insertion point */
12     inner_rem = outer_key_ptr/two ;          /* # inner-loop iterations remaining */
13         inner_key = $M$[inner_key_ptr] ;                    /* start 1st inner loop */
14         key_diff = inner_key − outer_key ;     /* if inner_key $\leq$ outer_key, then */
15         IF key_diff == 0 GOTO 30 ;            /* proceed to next inner iteration */
16         insert_key = outer_key + zero ;                      /* key to be inserted */
17         temp_ptr = outer_key_ptr + one;
18         insert_value = $M$[temp_ptr] ;                     /* value to be inserted */
19             temp_key = $M$[inner_key_ptr] ;              /* start of 2nd inner loop */
20             temp_ptr = inner_key_ptr + one;
21             temp_value = $M$[temp_ptr];
22             $M$[inner_key_ptr] = insert_key;
23             $M$[temp_ptr] = insert_value;
24             insert_key = temp_key + zero;
25             insert_value = temp_value + zero;
26             inner_key_ptr = inner_key_ptr + two;
27             IF inner_rem == 0 GOTO 7;
28             inner_rem = inner_rem − one;
29             IF zero == 0 GOTO 19;
30         inner_key_ptr = inner_key_ptr + two;
31         inner_rem = inner_rem − one;
32         IF inner_rem == 0 GOTO 7;
33         IF zero == 0 GOTO 13;
34 HALT ;                                                  /* not an actual command */
```

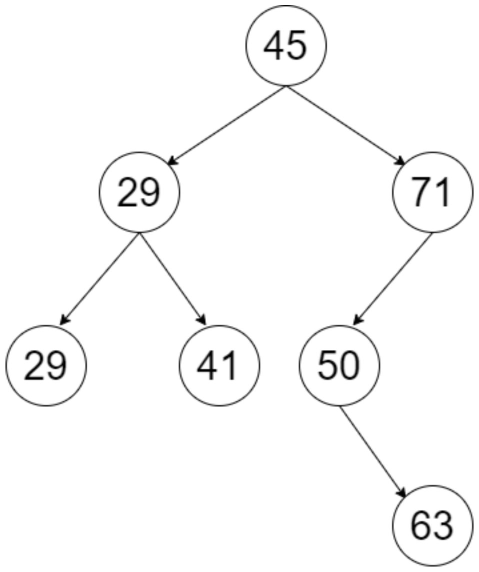
Algorithm 7.1: RAM implementation of `InsertionSort()`

8 Data

Implicit in the expressivity requirement is that we can describe the inputs and outputs of algorithms in the model. In the RAM model, all inputs and outputs are arrays of natural numbers. How can we represent other types of data?

- (Signed) integers:
- Rational numbers:
- Real numbers:
- Strings:

What about a fancy data structure like a binary search tree? We can represent a BST as an array of 4-tuples (K, V, P_L, P_R) where P_L and P_R are pointers to the left and right children. Let's consider the example from last class:



Assuming all of the associated values are 0, this would be represented as the following array of length 28:

$$[45, 0, 4, 8, 29, 0, 12, 16, 71, 0, 20, 0, 29, 0, 0, 0, 41, 0, 0, 0, 50, 0, 0, 24, 63, 0, 0, 0]$$

For nodes that do not have a left or right child, we assign the value of 0 to P_L or P_R. Assigning the pointer value to 0 does not refer to the value at the memory location of 0, as we assume that the root of the tree cannot be a child of any of the nodes in the rest of the tree. Note that there are many ways to construct a binary tree using this array representation.