Lecture Notes of CSCI5610 Advanced Data Structures

Yufei Tao
Department of Computer Science and Engineering
Chinese University of Hong Kong

January 7, 2020
Lecture 1: Course Overview and Computation Models

A data structure, in general, stores a set of elements, and supports certain operations on those elements. From your undergraduate courses, you should have learned two types of use of data structures:

- They alone can be employed directly for information retrieval (e.g., “find all the people whose ages are equal to 25”, or “report the number of people aged between 20 and 40”).
- They serve as building bricks in implementing algorithms efficiently (e.g., Dijkstra’s algorithm for finding shortest paths would be slow unless it uses an appropriate structure such as the priority queue).

This (graduate) course aims to deepen our knowledge of data structures. Specifically:

- We will study a number of new data structures for solving several important problems in computer science with strong performance guarantees (heuristic solutions, which perform well only on some inputs, may also be useful in some practical scenarios, but will not be of interest to us in this course).
- We will discuss a series of techniques for designing and analyzing data structures with non-trivial performance guarantees. Those techniques are generic in the sense that they are useful in a great variety of scenarios, and may very likely enable you to discover innovative structures in your own research.

Hopefully, with the above, you would be able to better appreciate the beauty of computer science at the end of the course.

The random access machine (RAM) model. Computer science is a subject under mathematics. From your undergraduate study, you should have learned that, before you can even start to analyze the “running time” of an algorithm, you need to first define a computation model properly.

Unless otherwise stated, we will be using the standard RAM model. In this model, the memory is an infinite sequence of cells, where each cell is a sequence of $w$ bits for some integer $w$, and is indexed by an integer address. Each cell is also called a word; and accordingly, the parameter $w$ is often referred to as the word length. The CPU, on the other hand, has a (constant) number of cells, each of which is called a register. The CPU can perform only the following atomic operations:

- Set a register to some constant, or to the content of another register.
- Compare two numbers in registers.
- Perform $+,-,\cdot,/\text{ on two numbers in registers.}$
- Shift the word in a register to the left (or right) by a certain number of bits.
• Perform the AND, OR, XOR on two registers.

• When an address $x$ has been stored in a register, read the content of the memory cell at address $x$ into a register, or conversely, write the content of a register into the memory cell.

The time (or cost) of an algorithm is measured by the number of atomic operations it performs. Note that the time is an integer.

A remark is in order about the word length $w$: it needs to be long enough to encode all the memory addresses! For example, if your algorithm uses $n^2$ memory cells for some integer $n$, then the word length will need to have at least $2 \log_2 n$ bits.

**Dealing with real numbers.** In the model defined earlier, the (memory/register) cells can only store integers. Next, we will slightly modify the model in order to deal with real values.

Note that simply “allowing” each cell to store a real value does not give us a satisfactory model because it creates several nasty issues. For example, how many bits would you use for a real value? In fact, even if the number of bits were infinite, still we would not be able to represent all the real values even in a short interval like $[0,1]$ — the set of real values in the interval is uncountably infinite! If we cannot even specify the word length for a “real-valued” cell, how to properly define the atomic operations for performing shifts and the logic operations AND, OR, and XOR?

We can alleviate this issue by introducing the concept of black box. We still allow a (memory/register) cell $c$ to store a real value $x$, but in this case, the algorithm is forbidden to look inside $c$, that is, the algorithm has no control over the representation of $x$. In other words, $c$ is now a black box, holding the value $x$ precisely (by magic).

A black box remains as a black box after computation. For example, suppose that two registers are both storing $\sqrt{2}$. We can calculate their product 2, but the product must still be understood as a real value (even though it is an integer). This is similar to the requirement in C++ that the product of two float numbers remains as a float number.

Now we can formally extend the RAM model as follows:

• Each cell can store either an integer or a real value.

• For operations $+,-,\ast,/$, if one of the operand numbers is a real value, the result is a real value.

• Among the atomic operations mentioned earlier, shifting, AND, OR, and XOR cannot be performed on registers that store real values.

We should note that, although mathematically sound, the resulting model — often referred to as the real RAM model — is not necessarily a realistic model in practice because no one has proven that it is polynomial-time equivalent to Turing machines (it would be surprising if it was). We must be very careful not to abuse the power of real value computation. For example, in the standard RAM model (with only integers), it is still open whether a polynomial time algorithm exists for the following problem:

**Input:** integers $x_1, x_2, \ldots, x_n$ and $k$

**Output:** whether $\sum_{i=1}^{n} \sqrt{x_i} \geq k$. 

2
It is rather common, however, to see people design algorithms by assuming that the square root operator can be carried out in polynomial time — in that case, the above problem can obviously be settled in polynomial time under the real-RAM model! We will exercise caution in the algorithms we design in this course, and will inject a discussion whenever issues like the above arise.

**Math conventions.** We will assume that you are familiar with the notations of $O(.)$, $\Omega(.)$, and $\Theta(.)$. We also use $\tilde{O}(f(n))$ to denote the class of functions that are $O(f(n) \cdot \text{polylog } n)$, namely, $\tilde{O}(.)$ hides a polylogarithmic factor. All the logarithms, unless explicitly stated otherwise, have base 2. $\mathbb{R}$ denotes the set of real values.
Lecture 2: The Binary Search Tree and The 2-3 Tree

This lecture will review the binary search tree (BST) which you should have learned from your undergraduate study. We will also talk about the 2-3 tree, which is a replacement of the BST that admits simpler analysis in proving certain properties. Both structures store a set \( S \) of elements conforming to a total order; for simplicity, we will assume that \( S \subseteq \mathbb{R} \). Set \( n = |S| \).

2.1 Binary search trees

2.1.1 The basics

A BST on \( S \) is a binary tree \( T \) satisfying the following properties:

- Every node \( u \) in \( T \) stores an element in \( S \), which is denoted as the key of \( u \). Conversely, every element in \( S \) is the key of exactly one node in \( T \). This means \( T \) has precisely \( n \) nodes.

- For every non-root node \( u \) with parent \( p \):
  - if \( u \) is the left child of \( p \), the keys stored in the subtree rooted at \( u \) are smaller than the key of \( p \);
  - if \( u \) is the right child of \( p \), the keys stored in the subtree rooted at \( u \) are larger than the key of \( p \).

The space consumption of \( T \) is clearly \( O(n) \) (cells). We say that \( T \) is balanced if its height is \( O(\log n) \). Henceforth, all BSTs are balanced unless otherwise stated.

The BST is a versatile structure that supports a large number of operations on \( S \) efficiently:

- **Insertion/deletion**: an element can be added to \( S \) or removed from \( S \) in \( O(\log n) \) time.

- **Predecessor/successor search**: the predecessor (or successor) of \( q \in \mathbb{R} \) is the largest (or smallest, resp.) element in \( S \) that is at most (or at least, resp.) \( q \). Given any \( q \), its predecessor/successor in \( S \) can be found in \( O(\log n) \) time.

- **Range reporting**: Given an interval \( I = [x, y] \) where \( x, y \in \mathbb{R} \), all the elements in \( I \cap S \) can be reported in \( O(\log n + k) \) time where \( k = |I \cap S| \).

- **Find-min/find-max**: Report the smallest/largest element of \( S \) in \( O(\log n) \) time.

The following are two more sophisticated operations that may not have been covered by your undergraduate courses:

- **Split**: Given a real value \( x \in S \), split \( S \) into two sets: (i) \( S_1 \) which includes all the elements in \( S \) less than \( x \), and (ii) \( S_2 = S \setminus S_1 \). Assuming a BST on \( S \), this operation also produces a BST on \( S_1 \) and a BST on \( S_2 \). All these can be done in \( O(\log n) \) time.
• **Join**: Given two sets $S_1$ and $S_2$ of real values such that $x < y$ for any $x \in S_1, y \in S_2$, merge them into $S = S_1 \cup S_2$. Assuming a BST on each of $S_1$ and $S_2$, this operation also produces a BST on $S$. All these can be done in $O(\log n)$ time.

It is a bit technical to implement the above two operations on the BST directly. This is the reason why we will talk about the 2-3 tree later (Section 2.2) which supports the two operations in an easier manner.

### 2.1.2 Slabs

Next we introduce the notion of *slab* which will appear very often in our discussion with BSTs.

Consider a BST $T$ on $S$. Let $u$ be a node in $T$ for which either the left child or the right child does not exist (note: $u$ is not necessarily a leaf node). In this case, we store a nil pointer for that missing child at $u$. It will be convenient to regard each nil pointer as a *conceptual leaf node*. You should not confuse this with a (genuine) leaf node $z$ of $T$ (every $z$ has two conceptual leaf nodes as its “children”). The total number of conceptual leaf nodes is exactly $n + 1$.

Given a node or a conceptual leaf node $u$ in $T$, we now define its slab, denoted as $slab(u)$, as follows:

- If $u$ is the root of $T$, $slab(u) = (-\infty, \infty)$.
- Otherwise, let the parent of $u$ be $p$, and $x$ the key of $p$. Now, proceed with:
  - if $u$ is the left child of $p$, then $slab(u) = slab(p) \cap (-\infty, x)$;
  - otherwise, $slab(u) = slab(p) \cap [x, \infty)$.

Note that $T$ defines exactly $2n + 1$ slabs.

**Example.** Figure 2.1 shows a BST on the set $S = \{10, 20, ..., 90\}$. The slab of node 40 is $[20, 50)$, while that of its right conceptual leaf is $[40, 50)$.

The following properties are easy to verify:

**Property 1:** For any node $u$ in $T$, $slab(u)$ covers all the keys stored in the subtree rooted at $u$.

**Property 2:** For any two nodes $u, v$ in $T$ (which may be conceptual leaf nodes):

- If $u$ is an ancestor of $v$, then $slab(v)$ is covered by $slab(u)$;
- If neither of the two nodes is an ancestor of the other, then $slab(u)$ is disjoint with $slab(v)$.
Property 3: The slabs of the $n+1$ conceptual leaf nodes partition $\mathbb{R}$.

Now we prove a very useful property:

**Lemma 1.** Any interval $q = [x, y)$, where $x$ and $y$ take values from $S$, $-\infty$, or $\infty$, can be partitioned into $O(\log n)$ disjoint slabs.

**Proof.** Let us first consider that $q$ has the form $[x, \infty)$. We can collect a set $\Sigma$ of disjoint slabs whose union equals $q$ as follows:

1. Initially, $\Sigma = \emptyset$, and set $u$ to the root of $T$.
2. If the key of $u$ equals $x$, then add slab$(u)$ to $\Sigma$, and stop.
3. If the key of $u$ is smaller than $x$, the set $u$ to the right child of $u$, and repeat from 2.
4. Otherwise, let $v$ be the right child of $u$ (note that $v$ may be a conceptual leaf). Add slab$(v)$ to $\Sigma$. Then, set $u$ to the left child of $u$, and repeat from 2.

Proving the lemma for general $q$ is left to you as an exercise.

Henceforth, we will refer to the slabs in the above lemma as the **canonical slabs** of $q$.

**Example.** In Figure 2.1, the interval $q = [30, 90)$ is partitioned by its canonical slabs $[30, 40)$, $[40, 50)$, $[50, 80)$, $[80, 90)$.

2.1.3 Augmenting a BST

The power of the BST can be further enhanced by associating its nodes with additional information. For example, we can store at each node $u$ of $T$ a count which is the number of keys stored at the subtree rooted at $u$. The resulting structure will be referred to as a **count BST** henceforth.

The count BST supports all the operations in Section 2.1 with the same performance guarantees. In addition, it also supports:

- **Range counting**: Given an interval $q = [x, y]$ with $x, y \in \mathbb{R}$, report $|q \cap S|$, namely, the number of elements in $S$ that are covered by $q$.

**Corollary 1.** A count BST supports the range counting operation in $O(\log n)$ time.

**Proof.** This is immediate from Lemma 1 (strictly speaking, the lemma requires the interval $q$ to be open on the right; how would you deal with this subtlety?).

2.2 The 2-3 tree

In a binary tree, every internal node has a fanout (i.e., number of child nodes) of either 1 or 2. We can relax this constraint by requiring only that each internal should have a constant fanout greater than 2. In this section, we will see a variant of the BST obtained following this idea. Interestingly, the variant, called the 2-3 tree, admits much simpler update (i.e., insertion/deletion) algorithms than the BST. We will explain how to support the split and join operations in Section 2.1.1 on the 2-3 tree.
2.2.1 The basics

A 2-3 tree on a set \( S \) of \( n \) real values is a tree \( T \) satisfying the following conditions:

- All the leaf nodes are at the same level (recall that the level of a node is the number of edges on its path to the root of \( T \)).
- Every internal node has 2 or 3 child nodes.
- Every leaf node \( u \) stores 2 or 3 elements in \( S \). The only exception arises when \( n = 1 \), in which case \( T \) has a single leaf node that stores the only element in \( S \).
- Every element in \( S \) is stored in a single leaf node.
- If an internal node \( u \) has child nodes \( v_1, ..., v_f \) where \( f = 2 \) or 3, it stores a routing element \( e_i \) for every child \( v_i \), which is the smallest element stored in the leaf nodes under \( v_i \).
- If an internal node \( u \) has child nodes \( v_1, ..., v_f \) (\( f = 2 \) or 3) with routing elements \( e_1, ..., e_f \), it must hold that all the elements stored at the leaf nodes under \( v_i \) are less than \( e_{i+1} \), for each \( i \in [1, f - 1] \).

Note that an element in \( S \) may be stored multiple times in the tree (definitely once in some leaf, but perhaps also as a routing element in some internal nodes). The height of \( T \) is \( O(\log n) \),

**Example.** Figure 2.2 shows a 2-3 tree on \( S = \{5, 12, 16, 27, 38, 44, 49, 63, 81, 87, 92, 96\} \). Note that the leaf nodes of the tree present a sorted order of \( S \). □

As a remark, if you are familiar with the B-tree, you can understand the 2-3 tree as a special case with \( B = 3 \).

2.2.2 Handling overflows and underflows

Assume that \( n \geq 2 \) (i.e., ignoring the special case where \( T \) has only a single element). An internal or leaf node overflows if it contains 4 elements, or underflows if it contains only 1 element.

**Treating overflows.** We consider the case where the overflowing node \( u \) is not the root of \( T \) (the opposite case is left to you). Suppose that \( u \) contains elements \( e_1, e_2, ..., e_4 \) in ascending order, and that \( p \) is the parent of \( u \). We create another node \( u' \), move \( e_3 \) and \( e_4 \) from \( u \) to \( u' \), and add a routing element \( e_3 \) to \( p \) for \( u' \). See Figure 2.3. The steps so far take in constant time. Note that at this moment \( p \) may be overflowing, which is then treated in the same manner. Since the overflow may propagate all the way to the root, in the worst case we spend \( O(\log n) \) time overall.
Treating underflows. We consider the case where the underflowing \( u \) is not the root of \( T \) (the opposite case is left to you). Suppose that the only element in \( u \) is \( e \), and that \( p \) is the parent of \( u \). Since \( p \) has at least two child nodes, \( u \) definitely has a sibling \( u' \); due to symmetry, we will discuss only the case where \( u' \) is the right sibling of \( u \). We proceed as follows:

- If \( u' \) has 2 elements, we move all the elements of \( u \) into \( u' \), delete \( u' \) from the tree, and remove the routing element in \( p \) for \( u' \). See Figure 2.4a. These steps require constant time. Note that \( p \) may be underflowing at this moment, which is treated in the same manner. Since the underflow may propagate all the way to the root, in the worst case we spend \( O(\log n) \) time overall.

- If \( u' \) has 3 elements \( e_1, e_2, e_3 \), in constant time we move \( e_1 \) from \( u' \) into \( u \), and modify the routing element in \( p \) for \( u' \). See Figure 2.4b. (Think: is there a chance the changes may propagate to the root?)

Remark. The underflow/overflow treating algorithms imply that an insertion or a deletion can be supported in \( O(\log n) \) time (why?).

2.2.3 Splits and joins

Recall that our main purpose for discussing the 2-3 tree is to seek a (relatively) easy way to support the split and join operations, re-stated below:
Join. Let us first deal with joins because the algorithm is simple, and will be leveraged to perform splits. Suppose that $T_1$ and $T_2$ are the 2-3 trees on $S_1$ and $S_2$, respectively. We can accomplish the join by adding one of the 2-3 trees as a subtree of the other. Specifically, denote by $h_1$ and $h_2$ the heights of $T_1$ and $T_2$, respectively. Due to symmetry, assume $h_1 \geq h_2$.

- If $h_1 = h_2$, just create a root $u$ which has $T_1$ as the left subtree and $T_2$ as the right subtree.
- Otherwise, set $\ell = h_1 - h_2$. Let $u$ be the level-$(\ell - 1)$ node on the rightmost path of $T_1$. Add $T_2$ as the rightmost subtree of $u$. See Figure 2.5. Note that this may trigger $u$ to overflow, which is then treated in the way explained earlier.

Overall, a join can be performed in $O(1 + \ell)$ time, which is $O(\log n)$.

Split. Due to symmetry, we will explain only how to produce the 2-3 tree of $S_1$. Let $\mathcal{T}$ be the 2-3 tree on $S$. First, find the path $\Pi$ in $\mathcal{T}$ from the root to the leaf containing the value $x$ (used for splitting). It suffices to focus on the part of $\mathcal{T}$ that is “on the left” of $\Pi$. Interestingly, this part can be partitioned into a set $\Sigma$ of $O(\log n)$ 2-3 trees. Before elaborating on this formally, let us first see an example.

Example. Consider Figure 2.6a where $\Pi$ is indicated by the bold edges. We can ignore subtrees labeled as IV and V because they are “on the right” of $\Pi$. Now, let us focus on the part “on the left” of $\Pi$. At the root $u_1$ (level 0), $\Pi$ descends from the 2nd routing element; the subtree labeled as I is added to $\Sigma$. At the level-1 node $u_2$, $\Pi$ descends from the 1st routing element; no tree is added to $\Sigma$. At the level-2 node $u_3$, $\Pi$ descends from the 3rd routing element; the 2-3 tree added to $\Sigma$ has $u_3$ as the root, but only two subtrees labeled as II and III, respectively. The same idea applies to every level. At the leaf level, what is added to $\Sigma$ is a 2-3 tree with only one node. Note how the 2-3 trees, shown in Figure 2.6b, together cover all the elements of $S_1$.

Formally, we generate $\Sigma$ by adding at most one 2-3 tree at each level $\ell$. Let $u$ be the level-$\ell$ node on $\Pi$. Denote by $e_1, \ldots, e_f$ the elements in $u$ where $f = 2$ or 3.
• If $\Pi$ descends from $e_1$, no tree is added to $\Sigma$.

• If $\Pi$ descends from $e_2$, we add the subtree referenced by $e_1$ to $\Sigma$.

• If $\Pi$ descends from $e_3$, we add the subtree rooted at $u$ to $\Sigma$, after removing $e_3$ and its subtree.

Denote by $T'_1, T'_2, \ldots, T'_t$ the 2-3 trees added by the above procedure in ascending order of level. Denote by $h_i$ the height of $T'_i$, $1 \leq i \leq t$. It must hold that:

$$h_1 \geq h_2 \geq \ldots \geq h_t.$$ 

We can now join all the trees together to obtain the 2-3 tree on $S_1$. To achieve $O(\log n)$ time, we must be careful with the order of joins. Specifically, we do the joins in descending order of $i$:

1. for $i = t$ to 2
2. $T'_{i-1} \leftarrow$ the join of $T'_{i-1}$ and $T'_i$

The final $T'_1$ is the 2-3 tree on $S_1$. The cost is all the joins is:

$$\sum_{i=1}^t O(1 + h_{i-1} - h_i) = O(t + h_1) = O(\log n).$$
Exercises

Problem 1. Complete the proof of Lemma 1.

Problem 2 (range max). Consider \( n \) people for each of whom we have her/his age and salary. Design a data structure of \( O(n) \) space to answer the following query in \( O(\log n) \) time: find the maximum salary of all the people aged between \( x \) and \( y \), where \( x, y \in \mathbb{R} \).

Problem 3. Let \( S \) is a set of \( n \) real values. Given a count BST on \( S \), explain how to answer following query in \( O(\log n) \) time: find the \( k \)-th largest element in \( S \), where \( k \) can be any integer from 1 to \( n \).

Problem 4. Let \( T \) be a 2-3 tree on a set \( S \) of \( n \) real values. Given any \( x \leq y \), describe an algorithm to obtain in \( O(\log n) \) time a 2-3 tree on the set \( S \setminus [x, y] \) (namely, the set of elements in \( S \) that are not covered by \([x, y]\)).

Problem 5* (meldable heap). Design a data structure of \( O(n) \) space to store a set \( S \) of \( n \) real values to satisfy the following requirements:

- An element can be inserted to \( S \) in \( O(\log n) \) time.
- The smallest element in \( S \) can be deleted in \( O(\log n) \) time.
- Let \( S_1, S_2 \) be two disjoint sets of real values. Given a data structure (that you have designed) on \( S_1 \) and another on \( S_2 \), you can obtain a data structure on \( S_1 \cup S_2 \) in \( O(\log(|S_1| + |S_2|)) \) time. Note that here we do not have the constraint that the values in \( S_2 \) should be larger than those in \( S_1 \).