Contents

1 Course Overview and Computation Models 3

2 The Binary Search Tree and the 2-3 Tree 6
   2.1 The binary search tree ........................................ 6
   2.2 The 2-3 tree .................................................... 8

3 Structures for Intervals 14
   3.1 The interval tree ............................................. 14
   3.2 The segment tree .............................................. 16

4 Structures for Points 18
   4.1 The kd-tree ................................................... 18
   4.2 A bootstrapping lemma ....................................... 20
   4.3 The priority search tree ..................................... 22
   4.4 The range tree ................................................. 25
   4.5 Another range tree with better query time ................. 27
   4.6 Pointer-machine structures .................................. 28

5 Logarithmic Method and Global Rebuilding 31
   5.1 Amortized update cost ........................................ 31
   5.2 Decomposable problems ....................................... 32
   5.3 The logarithmic method ....................................... 32
   5.4 Fully dynamic kd-trees with global rebuilding ............ 35

6 Weight Balancing 39
   6.1 BB[α]-trees .................................................. 39
   6.2 Insertion ....................................................... 40
   6.3 Deletion ....................................................... 40
   6.4 Amortized analysis .......................................... 40
   6.5 Dynamization with weight balancing ......................... 41
   6.6 Remarks ....................................................... 42

7 Partial Persistence 45
   7.1 The potential method ......................................... 45
   7.2 Partially persistent BST ..................................... 46
   7.3 General pointer-machine structures ......................... 49
8 Dynamic Perfect Hashing

8.1 Two random graph results .............................................. 52
8.2 Cuckoo hashing ........................................................... 53
8.3 Analysis ................................................................. 56
8.4 Remarks ................................................................. 57
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 $+$, $-$, $\cdot$, $/$ 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 +, −, ∗, /, 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 \( \sum_{i=1}^{n} \sqrt{x_i} \geq k \). |
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.

**Randomness.** All the atomic operations are deterministic so far. In other words, our models so far do not permit randomization, which is important to certain algorithmic techniques (such as hashing).

To fix the issue, we introduce one more atomic operation for both the RAM and real-RAM models. This operation, named $RAND$, takes two non-negative integer parameters $x$ and $y$, and returns an integer chosen uniformly at random from $[x, y]$. In other words, every integer in $[x, y]$ can be returned with probability $1/(y - x + 1)$. The values of $x, y$ should be in $[0, 2^w - 1]$ because they each need to be encoded in a word.

**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, $\tilde{O}(\cdot)$ 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 The binary search tree

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) $S_2 = S \setminus S$. 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 complicated 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 $n + 1$. Henceforth, we will use the term actual node to refer to a “genuine” node in $T$ that is not a conceptual leaf.

Given an actual/conceptual node $u$ in $T$, we now define its slab, denoted as $\text{slab}(u)$, as follows:

- If $u$ is the root of $T$, $\text{slab}(u) = (−\infty, \infty)$.
- 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 $\text{slab}(u) = \text{slab}(p) \cap (−\infty, x)$;
  - otherwise, $\text{slab}(u) = \text{slab}(p) \cap [x, \infty)$.

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 propositions are easy to verify:

**Proposition 2.1.** For any two nodes $u, v$ in $T$ (which may be actual or conceptual):

- If $u$ is an ancestor of $v$, then $\text{slab}(v)$ is covered by $\text{slab}(u)$;
- If neither of the two nodes is an ancestor of the other, then $\text{slab}(u)$ is disjoint with $\text{slab}(v)$. 

![Figure 2.1: A BST (every square is a conceptual leaf)](image-url)
Proposition 2.2. The slabs of the \( n + 1 \) conceptual leaf nodes partition \( \mathbb{R} \).

Now we prove a very useful property:

Lemma 2.1. 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 the slab of the right child of \( u \) (the child may be conceptual) 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, add the slab of the right child of \( u \) (the child may be conceptual) 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 2.1. A count BST supports the range counting operation in \( O(\log n) \) time.

Proof. This is immediate from Lemma 2.1 (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. This variant, called the 2-3 tree, is a replacement of the BST in the sense that it can essentially attain all the performance guarantees of the BST, but interestingly, often admits simpler analysis. We will explain how to support the split and join operations in Section 2.1.1 on the 2-3 tree (these operations can also be supported by the BST, but in a more complicated manner).
2.2.1 Description of the structure

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 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 2.4(a). 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 2.4(b). (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**: 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 2-3 tree on each of \( S_1 \) and \( S_2 \), this operation should also produce a 2-3 tree on \( S \). The time allowed is \( O(\log n) \).

**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 \( T \) be the 2-3 tree on \( S \). First, find the path \( \Pi \) in \( T \) from the root to the leaf containing the value \( x \) (used for splitting). It suffices to focus on the part of \( T \) that is “on the left” of \( \Pi \). Interestingly, this part can be partitioned into a set \( \Sigma \) of \( t = O(\log n) \) 2-3 trees. Before elaborating on this formally, let us first see an example.

**Example.** Consider Figure 2.6(a) 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 2.6(b), 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.

---

**Source:** Modified for clarity from [Laarhoven & van Rijswijk](https://www.is.titech.ac.jp/~laarhoven/ publications.html)
• 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 of 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 2.1.

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 \( S_1 \).
Lecture 3: Structures for Intervals

In this lecture, we will discuss the interval tree and the segment tree, which represent two different approaches to store a set $S$ of intervals of the form $\sigma = [x, y]$ where $x$ and $y$ are real values. The ideas behind the two approaches are very useful in designing sophisticated structures on intervals, segments, rectangles, etc. (this will be clear later in the course). Set $n = |S|$. The interval tree uses linear space (i.e., $O(n)$), whereas the segment tree uses $O(n \log n)$ space.

We will also take the chance to introduce the stabbing query. Formally, given a search value $q \in \mathbb{R}$, a stabbing query returns all the intervals $\sigma \in S$ satisfying $q \in \sigma$. Both the interval tree and the segment tree answer a query in $O(\log n + k)$ time, where $k$ is the number of intervals reported.

A remark is in order at this point. Since the interval tree has smaller space consumption, it may appear “better” than the segment tree. While there is some truth in this impression (e.g., the interval tree is indeed more superior for stabbing queries), we must bear in mind the real purpose of our discussion: to learn the approach that each structure takes to organize intervals. The segment tree approach may turn out to be more useful on certain problems, which we will see in the exercises.

3.1 The interval tree

3.1.1 Description of the structure

Given an interval $[x, y]$, we call $x$ and $y$ its left and right endpoints, respectively. Denote by $P$ the set of endpoints of the intervals in $S$.

To obtain an interval tree on $S$, first create a BST $T$ (Section 2.1) on $P$. For each node $u$ in $T$, define a set $\text{stab}(u)$ of intervals as follows:

$$\text{stab}(u) \text{ consists of every } \sigma \in S \text{ such that } u \text{ is the highest node in } T \text{ whose key is covered by } \sigma.$$  

We will refer to $\text{stab}(u)$ as the stabbing set of $u$. The intervals in $\text{stab}(u)$ are stored in two lists (i.e., two copies per interval): the first list sorts the intervals by left endpoint, while the second by right endpoint. Both lists are associated with $u$ (i.e., we store in $u$ a pointer to each list). This completes the construction of the interval tree.

**Example.** Consider $S = \{[1, 2], [3, 7], [4, 12], [5, 9], [6, 11], [8, 15], [10, 14], [13, 16]\}$. Figure 3.1 shows a BST on $P = \{1, 2, ..., 16\}$. The stabbing set of node 9 is $\{[6, 11], [4, 12], [5, 9], [8, 15]\}$; note that all the intervals in the stabbing set cover the key 9. The stabbing set of node 13, on the other hand, is $\{[10, 14], [13, 16]\}$.

It is easy to verify that every interval in $S$ belongs to the stabbing set of exactly one node. The space consumption of the interval tree is therefore $O(n)$.
Figure 3.1: An interval tree

3.1.2 Stabbing query

Let us see how to use the interval tree on $S$ constructed in Section 3.1.1 to answer a stabbing query search value $q$.

To get the main idea behind the algorithm, consider first the root $u$ of the BST $T$. Without loss of generality, let us assume that $q$ is less than the key $\kappa$ of $u$. We can forget about the intervals stored in the (stabbing sets of the nodes in the) right subtree of $u$, because all those intervals $[x, y]$ must satisfy $x \geq \kappa > q$, and hence, cannot cover $q$. We will, however, have to explore the left subtree of $u$, but that is something to be taken care of by recursion. At $u$, we must find a way to report the intervals in $\text{stab}(u)$ that cover $q$. Interestingly, this can be done in $O(1 + k_u)$ time, if $k_u$ intervals are reported in $\text{stab}(u)$. For this purpose, we utilize the fact that all the intervals $[x, y] \in \text{stab}(u)$ must contain $\kappa$. Therefore, $[x, y]$ contains $q$ if and only if $x \leq q$. We thus scan the intervals in $\text{stab}(u)$ in ascending order of left endpoint, and stop as soon as coming across an interval $[x, y]$ satisfying $x > q$.

This leads to the following algorithm for answering the stabbing query. First, descend a root-to-leaf path $\Pi$ of $T$ to reach the (only) conceptual leaf (Section 2.1.2) whose slab covers $q$. For every node $u$ on $\Pi$ with key $\kappa$:

- if $q < \kappa$, report the qualifying intervals in $\text{stab}(u)$ by scanning them in ascending order of left endpoint;
- if $q \geq \kappa$, report the qualifying intervals in $\text{stab}(u)$ by scanning them in descending order of right endpoint.

The query time is therefore

$$\sum_{u \in \Pi} O(1 + k_u) = O(\log n + k)$$

noticing that every interval is reported at exactly one node on $\Pi$. 
3.2 The segment tree

3.2.1 Description of the structure

As before, let $P$ be the set of end points in $S$. To obtain a segment tree on $S$, first create a BST $T$ on $P$. Recall from Lemma 2.1 that every interval $\sigma \in S$ can be divided into $O(\log n)$ canonical intervals, each of which is the slab of an actual/conceptual node $u$ in $T$. We assign $\sigma$ to every such $u$. Define $S_u$ as the set of all intervals assigned to $u$. We store $S_u$ in a linked list associated with $u$. This finishes the construction of the segment tree.

Example. Consider $S = \{[1, 2], [3, 7], [4, 12], [5, 9], [6, 11], [8, 15], [10, 14], [13, 16]\}$. Figure 3.2 shows a BST on $P = \{1, 2, ..., 16\}$ with the conceptual leaves indicated. Interval $[4, 12]$, for example, is partitioned into canonical intervals $[4, 5), [5, 9), [9, 11), [11, 12)$, and hence, is assigned to 4 nodes: the right conceptual leaf of node 4, node 7, node 10, and the left conceptual leaf of node 12. To illustrate $S_u$, let $u$ be node 10 in which case $S_u$ contains $[4, 12]$ and $[6, 11]$. Since every interval in $S$ has $O(\log n)$ copies, the total space of the segment tree is $O(n \log n)$.

3.2.2 Stabbing query

A stabbing query with search value $q$ can be answered with a very simple algorithm:

1. identify the set $\Pi$ of actual/conceptual nodes whose slabs contain $q$
2. for every node $u \in \Pi$
3. report $S_u$

Proposition 3.1. No interval is reported twice.

Proof. Follows from the fact that the canonical intervals of any interval are disjoint (Lemma 2.1).

Proposition 3.2. If $\sigma \in S$ covers $q$, $\sigma$ must have been reported.

Proof. Follows from the fact that one of the canonical intervals of $\sigma$ must cover $q$ (because they partition $\sigma$).

It is thus clear that the query cost is $O(\log n + k)$, noticing that $\Pi$ can be identified by following a single root-to-leaf path in $O(\log n)$ time.
### Exercises

**Problem 1.** Describe how to construct an interval tree on $n$ intervals in $O(n \log n)$ time.

**Problem 2.** Describe how to construct a segment tree on $n$ intervals in $O(n \log n)$ time.

**Problem 3.** Let $S$ be a set of $n$ intervals in $\mathbb{R}$. Design a structure of $O(n)$ space to answer the following query efficiently: given an interval $q = [x, y]$ in $\mathbb{R}$, report all the intervals $\sigma \in S$ such that $\sigma \cap q \neq \emptyset$. Your query time needs to be $O(\log n + k)$, where $k$ is the number of reported intervals.

**Problem 4 (stabbing max).** Suppose that we are managing a server. For every connection session to the server, we store its: (i) logon time, (ii) logoff time, and (iii) the network bandwidth consumed. Let $n$ be the number of sessions. Design a data structure of $O(n)$ space to answer the following query in $O(\log n)$ time: given a timestamp $t$, return the session with the largest consumed bandwidth among all the sessions that were active on $t$.

(Hint: even though the space needs to be $O(n)$, still organize the intervals following the approach of the segment tree.)

**Problem 5 (2D stabbing max).** Let $S$ be a set of $n$ axis-parallel rectangles in $\mathbb{R}^2$ (i.e., each rectangle in $S$ has the form $[x_1, x_2] \times [y_1, y_2]$). Each rectangle $r \in S$ is associated with a real-valued weight. Describe a structure of $O(n \log n)$ space that answers the following query in $O(\log^2 n)$ time: given a point $q \in \mathbb{R}^2$, report the maximum weight of the rectangles $r \in S$ satisfying $q \in r$.

**Problem 6.** Let $S$ be a set of $n$ horizontal segments of the form $[x_1, x_2] \times y$ in $\mathbb{R}^2$. Given a vertical segment $q = x \times [y_1, y_2]$, a query reports all the segments $\sigma \in S$ that intersect $q$. Design a data structure to store $S$ in $O(n \log n)$ space such that every query can be answered in $O(\log^2 n + k)$ time, where $k$ is the number of segments reported.
Lecture 4: Structures for Points

Each real value can be regarded as a 1D point; with this perspective, a BST can be regarded as a data structure managing 1D points. In this lecture, we will discuss several structures designed to manage multidimensional points in \( \mathbb{R}^d \) where the dimensionality \( d \geq 2 \) is a constant. Our discussion will focus on \( d = 2 \), while in the exercises you will be asked to obtain structures of higher dimensionalities by extending the ideas we will learn.

Central to our discussion is orthogonal range reporting. Let \( S \) be a set of points in \( \mathbb{R}^d \). Given an axis-parallel rectangle \( q = [x_1, y_1] \times [x_2, y_2] \times \ldots \times [x_d, y_d] \), an (orthogonal) range reporting query returns \( q \cap S \). This generalizes the 1D range reporting mentioned in Section 2.1.1 (which can be handled efficiently by a BST). Set \( n = |S| \). The structures to be presented in this lecture will provide different tradeoffs between space and query time.

For simplicity, we will assume that the points of \( S \) are in general position: no two points in \( S \) have the same x-coordinate or y-coordinate. This assumption allows us to focus on the most important ideas, and can be easily removed with standard tie breaking techniques, as we will see in an exercise.

4.1 The \( kd \)-tree

This data structure stores \( S \) in \( O(n) \) space, and answers a 2D range reporting query in \( O(\sqrt{n} + k) \) time, where \( k \) is the number of points in \( q \cap S \).

4.1.1 Structure

We describe the \( kd \)-tree in a recursive manner.

\( n = 1 \). If \( S \) has only a single point \( p \), the \( kd \)-tree has only a single node storing \( p \).

\( n \geq 2 \). Let \( \ell \) be a vertical line that divides \( P \) as evenly as possible, that is, there are at most \( \lceil n/2 \rceil \) points of \( P \) on each side of \( \ell \). Create a root node \( u \) of the \( kd \)-tree, and store \( \ell \) (i.e., the x-coordinate of \( \ell \)) at \( u \). Let \( P_1 \) (or \( P_2 \)) be the set of points in \( P \) that are on the left (or right, resp.) of \( \ell \). Let \( P_1 \) (or \( P_2 \)) be the set of points in \( P \) that are on the left (or right, resp.) of \( \ell \).

Consider now \( P_1 \). If \( |P_1| = 1 \), create a left child \( v_1 \) of \( u \) storing the only point in \( P_1 \). Next, we assume \( |P_1| \geq 2 \). Let \( \ell_1 \) be a horizontal line that divides \( P_1 \) as evenly as possible. Create a left child \( v_1 \) of \( u \) storing the line \( \ell_1 \). Let \( P_{11} \) (or \( P_{12} \)) be the set of points in \( P_1 \) that are below (or above, resp.) of \( \ell_1 \). Recursively, create a \( kd \)-tree \( T_{11} \) on \( P_{11} \) and a \( kd \)-tree \( T_{12} \) on \( P_{12} \). Make \( T_{11} \) and \( T_{12} \) the left and right subtrees of \( v_1 \), respectively.

The processing of \( P_2 \) is similar. If \( |P_2| = 1 \), create a right child \( v_2 \) of \( u \) storing the only point in \( P_2 \). Otherwise, let \( \ell_2 \) be a horizontal line that divides \( P_2 \) as evenly as possible. Create a right child \( v_2 \) of \( u \) storing the line \( \ell_2 \). Let \( P_{21} \) (or \( P_{22} \)) be the set of points in \( P_2 \) that are below (or above,
resp.) of $\ell_2$. Recursively, create a kd-tree $T_{21}$ on $P_{21}$ and a kd-tree $T_{22}$ on $P_{22}$. Make $T_{21}$ and $T_{22}$ the left and right subtrees of $v_2$, respectively.

The kd-tree is a binary tree where every internal node has two children, and that the points of $S$ are stored only at the leaf nodes. The total number of nodes is therefore $O(n)$.

For each node $u$ in the kd-tree, we store its minimum bounding rectangle (MBR) which is the smallest axis-parallel rectangle covering all the points stored in the subtree of $u$. Note that the MBR of an internal node $u$ can be obtained from those of its children in constant time.

**Example.** Figure 4.1 shows a kd-tree on a set $S$ of 12 points. The shaded rectangle illustrates the MBR of the node storing the horizontal line $\ell_3$ (i.e., the right child of the root).

4.1.2 Range reporting

Let $T$ be a kd-tree on $S$. A range reporting query can be answered by simply visiting all the nodes in $T$ whose MBRs intersect with the search rectangle $q$. Whenever a leaf node is encountered, we report the point $p$ stored there if $p \in q$.

Next, we will prove that the query cost is $O(\sqrt{n} + k)$. For this purpose, we divide the nodes $u$ accessed into two categories:

- Type 1: the MBR of $u$ intersects with a boundary edge of $q$ (note that $q$ has 4 boundary edges).
- Type 2: the MBR of $u$ is fully contained in $q$.

We will prove that there are $O(\sqrt{n})$ nodes of Type 1. In an exercise, you will be asked to prove that the number of nodes of Type 2 is bounded by $O(k)$. It will then follow that the query cost is $O(\sqrt{n} + k)$.

**Lemma 4.1.** Any vertical line $\ell$ can intersect with the MBRs of $O(\sqrt{n})$ nodes.

**Proof.** It suffices to prove the lemma only for the case where $n$ is a power of 2 (think: why?). Fix any $\ell$. We say that a node is $\ell$-intersecting if its MBR intersects with $\ell$. Let $f(n)$ be the maximum number of $\ell$-intersecting nodes in any kd-tree storing $n$ points. Clearly, $f(n) = O(1)$ for any constant $n$.

Now consider the kd-tree $T$ we constructed on $S$. Let $\hat{u}$ be the root of $T$; recall that $\hat{u}$ stores a vertical line $\ell_1$. Due to symmetry, let us assume that $\ell$ is on the right of $\ell_1$. Denote by $u$ the right
child of $p$; note that the line $\ell_2$ stored in $u$ is horizontal. Let $v_1$ and $v_2$ be the left and right child nodes of $u$, respectively. See Figure 4.2 for an illustration.

What can be the $\ell$-intersecting nodes in $T$? Clearly, they can only be $\hat{u}$, $u$, and the $\ell$-intersecting nodes in the subtrees of $v_1$ and $v_2$. Since the subtree of $v_1$ (or $v_2$) contains $n/4$ points, we thus have:

$$f(n) \leq 2 + 2 \cdot f(n/4).$$

Solving the recurrence gives $f(n) = O(\sqrt{n})$.

An analogous argument shows that any horizontal line can intersect with the MBR of $O(\sqrt{n})$ nodes, too. Observe that the MBR of any Type-1 node must intersect with at least one of the following 4 lines: the two vertical lines passing the left and right edges of $q$, and the two vertical lines passing the lower and upper edges of $q$. It thus follows that there can be $O(\sqrt{n})$ Type-1 nodes.

### 4.2 A bootstrapping lemma

This section will present a technique to obtain a structure that uses $O(n)$ space, and answers any range reporting query in $O(n^\epsilon + k)$ time, where $\epsilon > 0$ can be any small constant (for the kd-tree, $\epsilon = 1/2$). The core of our technique is the following lemma:

**Lemma 4.2.** Suppose that there is a structure $\Upsilon$ that can store $n$ points in $\mathbb{R}^2$ in at most $F(n)$ space, and answers a range reporting query in at most $Q(n) + O(k)$ time. For any integer $\lambda \in [2, n/2]$, there exists a structure that uses at most $\lambda \cdot F(\lceil n/\lambda \rceil) + O(n)$ space and answers a range reporting query in at most $2 \cdot Q(\lceil n/\lambda \rceil) + \lambda \cdot O(\log(n/\lambda)) + O(k)$ time.

**Proof.** Let $S$ be the set of $n$ points. Find $\lambda - 1$ vertical lines $\ell_1, \ldots, \ell_{\lambda-1}$ to satisfy the following requirements:

- No point of $S$ falls on any line.
- If $x_1, \ldots, x_{\lambda-1}$ are the x-coordinates of $\ell_1, \ldots, \ell_{\lambda-1}$, respectively, let us define $\lambda$ slabs as follows:
  - Slab 1 includes all the points of $\mathbb{R}^2$ with x-coordinate less than $x_1$;
  - Slab $i \in [2, \lambda - 1]$ includes all the points of $\mathbb{R}^2$ with x-coordinate in $[x_{i-1}, x_i)$;
  - Slab $\lambda$ includes all the points of $\mathbb{R}^2$ with x-coordinate at least $x_{\lambda-1}$.
We require that $S$ should have at most $\lceil n/\lambda \rceil$ points in each slab.

For each $i \in [1, \lambda]$, define $S_i$ to be the set of points in $S$ that are covered by slab $i$. For each $S_i$, we create two structures:

- The data structure $\mathcal{T}$ (as stated in the lemma) on $S_i$; we will denote the data structure as $T_i$.
- A BST $B_i$ on the y-coordinates of the points in $S_i$.

The space consumption is clearly $\lambda \cdot F(\lceil n/\lambda \rceil) + O(n)$.

Let us now discuss how to answer a range reporting query with search rectangle $q$. If $q$ falls entirely in some slab $i \in [1, \lambda]$, we answer the query using $T_i$ directly in $Q(\lceil n/\lambda \rceil) + O(k)$ time.

Consider now the case where $q$ intersects with at least two slabs. Denote by $q_i$ the intersection of $q$ with slab $i$, for every $i \in [1, \lambda]$. Each $q_i$ is one of the following types:

- Type 1: empty — this happens when $q$ is disjoint with slab $i$.
- Type 2: the x-range of $q_i$ is precisely the x-range of slab $i$ — this happens when the x-range of $q$ spans the x-range of slab $i$.
- Type 3: the x-range of $q_i$ is non-empty, but is shorter than that of slab $i$.

Figure 4.3 shows an example where $q$ is the shaded rectangle, and $\lambda = 6$. Rectangles $q_1$ and $q_6$ are of Type 1, $q_3$ and $q_4$ are of Type 2, while $q_2$ and $q_5$ are of Type 3.

For Type 1, we do not need to do anything. For Type 3, we deploy $T_i$ to find $q_i \cap S_i$ in $Q(\lceil n/\lambda \rceil) + O(k_i)$ time, where $k_i = |q_i \cap S_i|$. Note that there can be at most two rectangles of Type 3; so we spend at most $2 \cdot Q(\lceil n/\lambda \rceil) + O(k)$ time on them.

How about a rectangle $q_i$ of Type 2? A crucial observation is that we can forget about the x-dimension. Specifically, a point $p \in S_i$ falls in $q_i$ if and only if the y-coordinate of $p$ is covered by the y-range of $q_i$. We can therefore find all the points of $q_i \cap S_i$ using $B_i$ in $O(\log(n/\lambda) + k_i)$ time. Since there can be $\lambda$ rectangles of Type 2, we end up spending at most $\lambda \cdot O(\log(n/\lambda)) + O(k)$ time on them.

The above lemma is bootstrapping because once we have obtained a data structure for range reporting, it may allow us to improve ourselves “automatically”. For example, with the kd-tree, we
have already achieved $F(n) = O(n)$ and $Q(n) = O(\sqrt{n})$. Thus, by Lemma 4.2, for any $\lambda \in [2, n/2]$ we immediately have a structure of $\lambda \cdot F([n/\lambda]) = O(n)$ space whose query time is

$$O(\sqrt{n/\lambda}) + \lambda \cdot O(\log n)$$

plus the linear output time $O(k)$. Setting $\lambda$ to $\Theta(n^{1/3})$ makes the query time $O(n^{1/3} \log n + k)$; note that this is a polynomial improvement over the kd-tree!

But we can do even better! Now that we have achieved $F(n) = O(n)$ and $Q(n) = O(n^{1/3} \log n)$, for any $\lambda \in [2, n/2]$ Lemma 4.2 immediately yields another structure of $O(n)$ space whose query time is

$$\tilde{O}((n/\lambda)^{1/3}) + \lambda \cdot O(\log n)$$

plus the linear output time $O(k)$. Setting $\lambda$ to $\Theta(n^{1/4})$ makes the query time $\tilde{O}(n^{1/4}) + O(k)$, thus achieving another polynomial improvement!

Repeating this roughly $1/\epsilon$ times produces a structure of $O(n/\epsilon) = O(n)$ space and query time $O(n^{\epsilon} + k)$, where $\epsilon$ can be any positive constant.

### 4.3 The priority search tree

The 2D range reporting queries we have been considering so far are 4-sided because the query rectangle $q$ is “bounded” on all sides. More specifically, if we write $q$ as $[x_1, x_2] \times [y_1, y_2]$, all the four values $x_1, x_2, y_1,$ and $y_2$ are finite (they are neither $\infty$ nor $-\infty$). Such queries are difficult in the sense that no linear-size structures known today are able to guarantee a query time of $O(\log n + k)$.

If exactly one of the four values $x_1, x_2, y_1,$ and $y_2$ takes an infinity value (i.e., $-\infty$ or $\infty$), $q$ is said to be 3-sided. More specially, if (i) two of the four values $x_1, x_2, y_1,$ and $y_2$ take infinity values, and (ii) they are on different dimensions, $q$ is said to be 2-sided. See Figure 4.4 for an illustration.

Clearly, 3-sided queries are special 4-sided queries. Therefore, a structure on 4-sided queries also works on 3-sided queries, and but not the vice versa. In this section, we will introduce a 3-sided structure called the priority search tree which uses linear space, and answers a (3-sided) query in $O(\log n + k)$ time, where $k$ is the number of points reported. Note that this is significantly better than using a kd-tree to answer 3-sided queries. The new structure also works on 2-sided queries because they are special 3-sided queries.

Due to symmetry, we consider search rectangles of the form $q = [x_1, x_2] \times [y, \infty)$ (as shown in the middle of Figure 4.4).
4.3.1 Structure

To create a priority search tree on $S$, first create a BST $T$ on the x-coordinates of the points in $S$. Each actual/conceptual node $u$ in $T$ may store a pilot point, defined recursively as follows:

- If $u$ is the root of $T$, its pilot point is the highest point in $S$.
- Otherwise, its pilot point is the highest among those points $p$ satisfying
  - the x-coordinate of $p$ is in slab($u$) (see Section 2.1.2 for the definition of slab), and
  - $p$ is not the pilot point of any proper ancestor of $u$.

If no such point exists, $u$ has no pilot point associated.

This finishes the construction of the priority search tree. Note that every point in $S$ is the pilot point of exactly one node (which is possibly conceptual). It is clear that the space is $O(n)$.

Example. Figure 4.5 shows a priority search tree on the point set \{a, b, ..., l\}. The x-coordinate of a point $p$ is denoted as $x_p$ in the tree.

Remark. Observe that the priority search tree is simultaneously a max heap on the y-coordinates of the points in $S$. For this purpose, the priority search tree is also known by the name treap.

4.3.2 Answering a 3-sided query

Before talking about general 3-sided queries, let us first consider a (very) special version: the search rectangle $q$ has the form $(-\infty, \infty) \times [y, \infty)$ (namely, $q$ is “1-sided”). Equivalently, this is to ask how we can use the priority search tree to efficiently report all the points in $S$ whose y-coordinate are at least $y$. Phrased yet in another way, this is to ask how we can we efficiently find all the keys at least $y$ in a max heap. This can be done in $O(1 + k)$ time, where $k$ is the number of elements returned.

Lemma 4.3. Given a search rectangle $q = (-\infty, \infty) \times [y, \infty)$, we can find all the points in $S \cap q$ in $O(1 + k)$ time, where $k = |S \cap q|$.

Proof. We answer the query using the following algorithm (setting $u$ to the root of $T$ initially):
Figure 4.6: Search paths \( \Pi_1 \) and \( \Pi_2 \) and the portion in between

```plaintext
\textbf{report-subtree}(u, y)
/* u is an actual/conceptual node in }T^*/
1. if \( u \) has no pilot point or its pilot point \( p \) has \( y \)-coordinate < \( y \) then return
2. report \( p \)
3. if \( u \) is a conceptual leaf then return
4. \textbf{report-subtree}(v_1, y) where \( v_1 \) is the left child of \( u \) (\( v_1 \) is possibly conceptual)
5. \textbf{report-subtree}(v_2, y) where \( v_2 \) is the right child of \( u \) (\( v_2 \) is possibly conceptual)
```

The correctness follows from the fact that the pilot point of \( u \) is the highest among all the pilot points stored in the subtree of \( u \).

To bound the cost, notice that each node \( u \) we access can be divided into two types:

- Type 1: the pilot point of \( u \) is reported.
- Type 2: the pilot point is not reported.

Clearly, there are at most \( k \) nodes of Type 1. How many nodes of Type 2? A crucial observation is that the parent of a type-2 node must be of Type 1. Therefore, there can be at most \( 2k \) nodes of Type 2. The total cost is therefore \( O(1 + k) \).

**Example.** Suppose that \( q \) is the shaded region as shown in Figure 4.5 (note that \( q \) has the form \((-\infty, \infty) \times [y, \infty)\)). The nodes accessed are: \( x_e, x_l, x_a, x_i, x_d, x_g, x_k, x_c, x_h, \) and \( x_f \).

We are now ready to explain how to answer a general 3-sided query with \( q = [x_1, x_2] \times [y, \infty) \). Without loss of generality, we can assume that \( x_1 \) and \( x_2 \) are the \( x \)-coordinated of some points in \( S \) (think: why?). Let us first find

- the path \( \Pi_1 \) in \( T \) from the root to the node storing the \( x \)-coordinate \( x_1 \);
- the path \( \Pi_2 \) in \( T \) from the root to the node storing the \( x \)-coordinate \( x_2 \).

Figure 4.6 illustrates how \( \Pi_1 \) and \( \Pi_2 \) look like in general: they descend from the root and diverge at some node. We are interested in only the nodes \( u \) that

- are in \( \Pi_1 \cup \Pi_2 \), or
- satisfy \( \text{slab}(u) \subseteq [x_1, x_2] \) — such are nodes are “in-between” \( \Pi_1 \) and \( \Pi_2 \) (the shaded portion in Figure 4.6).
For every other node $v$ (violating both of the above), $slab(v)$ must be disjoint with $[x_1, x_2]$; and therefore, the pilot point $p$ of $v$ cannot fall in $q$ (recall that $slab(v)$ must cover the x-coordinate of $p$).

This gives rise to the following the query algorithm:

1. find the paths $\Pi_1, \Pi_2$ as described above
2. for every node $u \in \Pi_1 \cup \Pi_2$
3. report the pilot point $p$ of $u$ if $p \in q$
4. find the set $\Sigma$ of actual/conceptual nodes whose slabs are the canonical slabs of $[x_1, x_2]$
5. for every node $u \in \Sigma$
6. report-subtree$(u, y)$

For every node $u \in \Sigma$, Line 5 finds all the qualifying pilot points (i.e., covered by $q$) that are stored in the subtree rooted at $u$, because (i) the subtree itself is a max heap, and (ii) we can forget about the x-range $[x_1, x_2]$ of $q$ in exploring the subtree of $u$. By Lemma 4.3, the cost of report-subtree$(u, y)$ is $O(1 + k_u)$ where $k_u$ is the number of points reported from the subtree of $u$.

The total query cost is therefore bounded by

$$O \left( |\Pi_1| + |\Pi_2| + \sum_{u \in \Sigma} (1 + k_u) \right) = O(\log n + k).$$

The filtering technique. Usually when we look at a query time complexity such as $O(\log n + k)$, we would often interpret the $O(\log n)$ term as the “search time we are prepared to waste without reporting anything”, and the $O(k)$ term as the “reporting time we are justified to pay”. For example, in using a BST to answer a 1D range reporting query, we may waste $O(\log n)$ time because there can be $O(\log n)$ nodes that need to be visited but contribute nothing to the query result. As another example, in using a kd-tree to answer a 2D (4-sided) range reporting query, the number of such nodes is $O(\sqrt{n})$.

The above interpretation, however, misses a very interesting point: we can regard $O(\log n + k)$ more generally as $O(\log n + k + k)$, which says that we can actually “waste” as much as $O(\log n + k)$ time in “searching”! Indeed, this is true for using the priority search tree to answer a 3-sided query: notice that the algorithm may access $O(\log n + k)$ nodes whose pilot points are not reported! Subtly, we charge the time “wasted” this way on the output. Only after we have reported the pilot point $p$ of a node $u$ will we search the child nodes of $u$. The $O(1)$ cost of searching the child nodes hence is “paid” for by the reporting of $p$.

This idea (of charging the search time on the output) is known as the filtering technique.

4.4 The range tree

We now return to 4-sided queries, i.e., the search rectangle $q$ is an arbitrary axis-parallel rectangle. We will introduce the range tree which consumes $O(n \log n)$ space, and answers a query in $O(\log^2 n + k)$ time.

4.4.1 Structure

First create a BST $T$ on the x-coordinates of the points in $S$. For each actual/conceptual node $u$ in $T$, denote by $S_u$ the set of points $p \in S$ satisfying $x_p \in slab(u)$ (recall that $x_p$ is the x-coordinate of
p). For every node $u$, we associate it with a secondary BST $T_u'$ on the y-coordinates of the points in $S_u$. Every point $p \in S_u$ is stored at the node in $T_u'$ corresponding to the y-coordinate $y_p$ of $p$.

Example. Figure 4.7 shows the BST $T$ for the set of points shown on the left of the figure. If $u$ is the node $x_l$, $S_u = \{i, a, d, l, g, b\}$. The secondary BST of $u$ is created on the y-coordinates of those points. Point $b$ is stored in the secondary BSTs of the right conceptual child of node $x_b$, node $x_b$ itself, node $x_g$, node $x_l$, and node $x_e$.  

Proposition 4.1. For each point $p \in S$, $x_p$ appears in the slabs of $O(\log n)$ nodes.

Proof. By Proposition 2.1, if the slabs of two nodes $u, v$ in $T$ intersect, one of $u, v$ must be an ancestor of the other. Thus, all the nodes whose slabs contain $x_p$ must be on a single root-to-leaf path in $T$. The proposition follows from the fact that the height of $T$ is $O(\log n)$.

The space consumption is therefore $O(n \log n)$.

4.4.2 Range reporting

We answer a range reporting query with search rectangle $q = [x_1, x_2] \times [y_1, y_2]$ as follows (assuming $x_1$ and $x_2$ are the x-coordinates of some points in $S$, without loss of generality):

1. find the set $\Sigma$ of nodes in $T$ whose slabs are the canonical slabs of $[x_1, x_2]$
2. for each node $u \in \Sigma$
   3. use $T_u'$ to report $\{p \in S_u \mid y_p \in [y_1, y_2]\}$

Proposition 4.2. Every point $p$ in $q \cap S$ is reported exactly once.

Proof. Clearly, $x_p \in [x_1, x_2]$. Therefore, $x_p$ appears in exactly a canonical slab of $[x_1, x_2]$ (by Lemma 2.1, the canonical slabs form a partition of $[x_1, x_2]$). Let $u$ be the node whose slab($u$) is that canonical slab. Thus, $p \in S_u$ and will be reported only there.

The proof of the next proposition is left to you as an exercise:

Proposition 4.3. The query time is $O(\log^2 n + k)$.  

26
4.5 Another range tree with better query time

In this section, we will present a data structure that (finally) answers a 4-sided query in $O(\log n + k)$ time, while still retaining the $O(n \log n)$ space complexity. This is achieved by combining the range-tree idea with the priority search tree (Section 4.3), and converting a 4-sided query to two 3-sided queries.

4.5.1 Structure

First create a BST $T$ on the x-coordinates of the points in $S$. For each actual/conceptual node $u$ in $T$, denote by $S_u$ the set of points $p \in S$ satisfying $x_p \in \text{slab}(u)$. For every actual node $u$ with key $\kappa$, define:

- $S_u^\leq$: the set of points $p \in S_u$ whose x-coordinate is less than $\kappa$;
- $S_u^\geq$: the set of points $p \in S_u$ whose x-coordinate is at least $\kappa$.

Note that $S_u^\leq$ and $S_u^\geq$ partition $S_u$. We associate $u$ with two secondary structures:

- $\sqsubseteq u$: a priority search tree on $S_u^\leq$ to answer “right-open” 3-sided queries, i.e., with search rectangles of the form $[x, \infty) \times [y_1, y_2]$;
- $\sqsupseteq u$: a priority search tree on $S_u^\geq$ to answer “left-open” 3-sided queries, i.e., with search rectangles of the form $(-\infty, x] \times [y_1, y_2]$.

The space is $O(n \log n)$ by Proposition 4.1 (recall that each priority search tree uses space linear to the number of points stored).

**Example.** In Figure 4.8, as an example, let $u$ be the node $x_l$. $\sqsubseteq u$ is created on $S_u^\leq = \{a, d, i\}$, while $\sqsupseteq u$ on $S_u^\geq = \{b, l, g\}$. 

4.5.2 Range reporting

Given a query with search rectangle $q = [x_1, x_2] \times [y_1, y_2]$ (assuming $x_1$ and $x_2$ are the x-coordinates of some points in $S$, without loss of generality), we answer it at the highest node $u$ in $T$ whose key $\kappa$ is covered by $q$. Specifically, we construct

$$q_{\leq} = [x_1, \infty) \times [y_1, y_2]$$
$$q_{\geq} = (-\infty, x_2] \times [y_1, y_2]$$
It is easy to see that
\[
q \cap S = (q \cap S_u^<) \cup (q \cap S_u^>).
\] (4.1)

**Example.** Consider that search rectangle \( q \) shown in Figure 4.8. Node \( x_l \) is the node \( u \) designated earlier. Note how \( q \) is decomposed into two 3-sided rectangles: \( q^< \) is the part colored in gray, while \( q^> \) the part in white. The 3-sided query \( q^< \) on \( S_u^< \) returns \( \{d\} \), while the 3-sided query \( q^> \) on \( S_u^> \) returns \( \{g\} \). The union of \( \{d\} \) and \( \{g\} \) gives the result of the 4-sided query \( q \). □

Using the priority search trees \( \sqsubseteq u \) and \( \sqsupseteq u \), the point set on the right side of (4.1) can be retrieved in \( O(\log n + k) \) time.

### 4.6 Pointer-machine structures

Have you noticed that, in all the structures we have discussed so far, the exploration of their content is always performed by following pointers? Indeed, they belong to a general class of structures known as the **pointer machine class**.

Formally, a **pointer machine structure** is a directed graph \( G \) satisfying the following conditions:

- There is a special node \( r \) in \( G \) that is called the **root**.
- Every node in \( G \) stores a constant number of words.
- Every node in \( G \) has a constant number of outgoing edges (but may have an arbitrary number of incoming edges).
- Any algorithm that accesses \( G \) must follow the rules below:
  - The first node visited must be the root \( r \).
  - The algorithm is permitted to access a non-root node \( u \) in \( G \) only if it has already accessed an in-neighbor of \( u \). This implies that the algorithm must have found a path from \( r \) to \( u \) in \( G \).

You may convince yourself that all our structures so far, as well as their accompanying algorithms, satisfy the above conditions.

One simple structure that is **not** in the pointer machine class is the array. Recall that, given an array \( A \) of \( n \), we can access directly \( A[i] \) for any \( i \in [1, n] \) in constant time, without following any path from some “root”.

Pointer-machine structures bear unique importance in computer science because they are applicable in scenarios where it is not possible to perform any (meaningful) calculation on **addresses**. One such scenario arises from distributed computing where each “node” is a light weighted machine (e.g., your cell phone). A pointer to a node \( u \) is the IP address of machine \( u \). No “arrays” can be implemented in such a scenario because, to enable constant time access to \( A[i] \), you need to calculate the address of \( A[i] \) by adding \( i \) to the starting address of \( A \) — something not possible in distributed computing (adding \( i \) to an IP address tells you essentially nothing).

Range reporting on pointer machines has been well understood. In 2D space, any pointer-machine structures achieving \( O(\text{polylog } n + k) \) query time — let alone \( O(\log n + k) \) — must consume
\(\Omega(n^{\log \frac{n}{\log \log n}})\) space \([7]\). A structure matching this lower bound and attaining \(O(\log n + k)\) query time has been found \([5]\). Note that the our structure in Section 4.5 is nearly optimal, except that its space is higher than the lower bound by an \(O(\log \log n)\) factor. Similar results also hold for higher dimensionalities, except that both the space and query complexities increase by \(O(\text{polylog } n)\) factors; see \([1,7]\).

By fully leveraging the power of the RAM model (address calculation and atomic operations that manipulate the bits within a word), it is possible to design structures with better complexities outside the pointer-machine class. For example, in 2D space, it is possible to achieve \(O(\log n + k)\) time using \(O(n \log^\epsilon n)\) space, where \(\epsilon > 0\) can be any small constant \([2,6]\). See also \([4]\) for results of higher dimensionalities.
Exercises

Problem 1. Prove that there can be $O(k)$ nodes of Type 2 (as defined in Section 4.1.2).

Problem 2. Describe an algorithm to build the kd-tree on $n$ points in $O(n \log n)$ time.

Problem 3. Explain how to remove the general position assumption for the kd-tree. That is, you still need to retain the same space and query complexities even if the assumption does not hold.

Problem 4. Let $S$ be a set of points in $\mathbb{R}^d$ where $d \geq 2$ is a constant. Extend the kd-tree to obtain a structure of $O(n)$ space that answers any $d$-dimensional range reporting query in $O(n^{1-1/d} + k)$ time, where $k$ is the number of points reported.

Problem 5. What is the counterpart of Lemma 4.2 in 3D space?

Problem 6*. Improve the query time in Lemma 4.2 to $2 \cdot Q(\lceil n/\lambda \rceil) + O(\log n + \lambda + k)$.

(Hint: one way to do so is to use the interval tree and stabbing queries.)

Problem 7. Consider the stabbing query discussed in Lecture 3 on a set $S$ of $n$ intervals in $\mathbb{R}$. Show that you can store $S$ in a priority search tree such that any stabbing query can be answered in $O(\log n + k)$ time, where $k$ is the number of intervals reported.

(Hint: turn the query into a 2-sided range reporting query on a set of $n$ points converted from $S$.)

Problem 8. Prove Proposition 4.3.

Problem 9. Let $S$ be a set of points in $\mathbb{R}^d$ where $d$ is a constant. Design a data structure that stores $S$ in $O(n \log^{d-1} n)$ space, and answers any orthogonal range reporting query on $S$ in $O(\log^{d-1} n + k)$ time, where $k$ is the number of reported points.

Problem 10 (range counting). Let $S$ be a set of $n$ points in $\mathbb{R}^2$. Given an axis-parallel rectangle $q$, a range count query reports $|q \cap S|$, i.e., the number of points in $S$ that are covered by $q$. Design a structure that stores $S$ in $O(n \log n)$ space, and answers a range count query in $O(\log^2 n)$ time.

Problem 11*. Let $S$ be a set of $n$ horizontal segments of the form $[x_1, x_2] \times y$ in $\mathbb{R}^2$. Given a vertical segment $q = x \times [y_1, y_2]$, a query reports all the segments $\sigma \in S$ that intersect $q$. Design a data structure to store $S$ in $O(n)$ space such that every query can be answered in $O(\log^2 n + k)$ time, where $k$ is the number of segments reported. (This improves an exercise in Lecture 3.)

(Hint: use the interval tree as the base tree, and the priority search tree as secondary structures.)

Problem 12. Prove: on a pointer-machine structure $G$ with $n$ nodes, the longest path from the root to a node in $G$ has length $\Omega(\log n)$. (This implies that $O(\log n + k)$ is the best query bound one can hope for range reporting using pointer-machine structures.)

(Hint: suppose that each node has an outdegree of 2. Starting from the root, how many nodes can you reach within $x$ hops?)
Lecture 5: Logarithmic Method and Global Rebuilding

We have seen some interesting data structures so far, but there is an issue: they are all *static* (except the BST and the 2-3 tree). It is not clear how they can be updated when the underlying set $S$ of elements undergoes changes, i.e., insertions and deletions. This is something we will fix in the next few lectures.

In general, a structure is *semi-dynamic* if it allows elements to be inserted but not deleted; it is (fully) *dynamic* if both insertions and deletions are allowed. In this lecture, we will learn a powerful technique called the *logarithmic method* for turning a static structure into a semi-dynamic one. The technique is generic because it works (in exactly the same way) on a great variety of structures.

We will use the kd-tree (Section 4.1) to illustrate the technique. Indeed, the kd-tree serves as an excellent example because it seems exceedingly difficult to support any updates on that structure. Several constraints must be enforced for the structure to work. For example, the first cut ought to be a vertical line $\ell$ that divides the input set of points *as evenly as possible*. Unfortunately, a single point insertion would throw off the balance and thus destroy the whole tree. It may therefore be surprising that later we will make the kd-tree semi-dynamic *without* changing the structure at all!

There is another reason why we want to discuss the kd-tree: it can actually support deletions in a fairly easy way! In general, if a structure can support deletions but not insertions, the logarithmic method would turn it into a fully dynamic structure. Sometimes, for that to happen, we would also need to perform *global rebuilding*, which simply rebuilds everything from scratch! This is also something that can be illustrated very well by the kd-tree.

### 5.1 Amortized update cost

Recall that the BST supports an update (i.e., insertion/deletion) in $O(\log n)$ *worst-case* time. This means that an update definitely finishes after $O(\log n)$ atomic operations, where $n$ is the number of nodes in the BST *currently* (assuming $n \geq 2$).

In this course, we will *not* aim to achieve worst-case update time (see, however, the remark at the end of this subsection). Instead, we will focus on obtaining small *amortized* update time. But what does it mean exactly to say that a structure has amortized time, for example, $O(\log n)$?

A structure with low amortized update cost should be able to support any number of updates with a small *total* cost, even though the time of an individual update may be large. Formally:

- a structure supports an insertion in $U_{ins}$ *amortized* time if it can process any sequence of $t$ insertions in $t \cdot U_{ins}$ time;
- a structure supports a deletion in $U_{del}$ *amortized* time if it can process any sequence of $t$ deletions in $t \cdot U_{del}$ time;
• a structure supports an insertion in $U_{ins}$ amortized time and simultaneously a deletion in $U_{del}$ amortized time if it can process any update sequence containing $t_{ins}$ insertions and $t_{del}$ deletions (in an arbitrary order) in $t_{ins} \cdot U_{ins} + t_{del} \cdot U_{del}$ time.

Remark. There are standard de-amortization techniques (see [11]) that convert a structure with small amortized update time into a structure with small worst case update time. Therefore, for many problems, it suffices to focus on amortized cost. The curious students may approach the instructor for a discussion.

5.2 Decomposable problems

We have discussed many types of queries, each of which retrieves certain information about the elements in the input set satisfying some conditions specified by the query. For example, for range reporting, the “information” is simply the elements themselves, whereas for range counting (Section 2.1.3), it is the number of those elements.

We say that a query is decomposable if the following is true for any disjoint sets of elements $S_1$ and $S_2$: given the query answer on $S_1$ and the answer on $S_2$, the answer on $S_1 \cup S_2$ can be obtained in constant time.

Consider, for example, orthogonal range reporting on 2D points. Given an axis-parallel rectangle $q$, the query answer on $S_1$ (or $S_2$) is the set $\Sigma_1$ (or $\Sigma_2$) of points therein covered by $q$. Clearly, $\Sigma_1 \cup \Sigma_2$ is the answer of the same query on $S_1 \cup S_2$. In other words, once $\Sigma_1$ and $\Sigma_2$ are available, we have already obtained the answer on $S_1 \cup S_2$ (nothing needs to be done). Hence, the query is decomposable.

As another example, consider range counting on a set of real values. Given an interval $q \subseteq \mathbb{R}$, the query answer on $S_1$ (or $S_2$) is the number $c_1$ (or $c_2$) of values therein covered by $q$. Clearly, $c_1 + c_2$ is the answer of the same query on $S_1 \cup S_2$. In other words, once $c_1$ and $c_2$ are available, we can obtain the answer on $S_1 \cup S_2$ in constant time. Hence, the query is decomposable.

Verify for yourself that all the queries we have seen so far are decomposable: predecessor/successor, find-min/max, range reporting, range counting/max, stabbing, etc.

5.3 The logarithmic method

This section serves as a proof of the following theorem:
Theorem 5.1. Suppose that there is a static structure $\mathcal{Y}$ that
- stores $n$ elements in at most $F(n)$ space;
- can be constructed in at most $n \cdot U(n)$ time;
- answers a decomposable query in at most $Q(n)$ time (plus, if necessary, a cost linear to the number of reported elements).

Set $h = \lceil \log_2 n \rceil$. There is a semi-dynamic structure $\mathcal{Y}'$ that
- stores $n$ elements in at most $\sum_{i=0}^{h} F(2^i)$ space;
- supports an insertion in $O \left( \sum_{i=0}^{h} U(2^i) \right)$ amortized time;
- answers a decomposable query in $O(\log n) + \sum_{i=0}^{h} Q(2^i)$ time (plus, if necessary, a cost linear to the number of reported elements).

Before proving the theorem, let us first see its application on the kd-tree. We know that the kd-tree consumes $O(n)$ space, can be constructed in $O(n \log n)$ time (this was an exercise of Lecture 4), and answers a range reporting query in $O(\sqrt{n} + k)$ time, where $k$ is the number of reported elements. Therefore:

\[
F(n) = O(n)
\]
\[
U(n) = O(\log n)
\]
\[
Q(n) = O(\sqrt{n}).
\]

Theorem 5.1 immediately gives a semi-dynamic structure that uses

\[
\sum_{i=0}^{\lceil \log_2 n \rceil} O(2^i) = O(n)
\]
space, supports an insertion in

\[
\sum_{i=0}^{\lceil \log_2 n \rceil} O(\log 2^i) = O(\log^2 n)
\]
time, and answers a query in

\[
\sum_{i=0}^{\lceil \log_2 n \rceil} O(\sqrt{2^i}) = O(\sqrt{n})
\]
plus $O(k)$ time.

5.3.1 Structure

Let $S$ be the input set of elements; set $n = |S|$ and $h = \lceil \log_2 n \rceil$. At all times, we divide $S$ into disjoint subsets $S_0, S_1, ..., S_h$ (some of which may be empty) satisfying:

\[
|S_i| \leq 2^i.
\]

Create a structure of $\mathcal{Y}$ on each subset; denote by $\mathcal{Y}_i$ the structure on $S_i$. Then, $\mathcal{Y}_1, \mathcal{Y}_2, ..., \mathcal{Y}_h$ together constitute our “overall structure”. The space usage is bounded by $\sum_{i=0}^{h} F(2^i)$.

Remark. At the beginning when we construct our “overall structure” from scratch, it suffices to set $S_h = S$, and $S_i = \emptyset$ for every $i \in [0, h - 1]$. 

33
5.3.2 Query

To answer a query \( q \), we simply search all of \( \Upsilon_1, \ldots, \Upsilon_h \). Since the query is decomposable, we can obtain the answer on \( S \) from the answers on \( S_1, \ldots, S_h \) in \( O(h) \) time. The overall query time is therefore

\[
O(h) + \sum_{i=0}^{h} Q(2^i) = O(\log n) + \sum_{i=0}^{h} Q(2^i).
\]

5.3.3 Insertion

To insert an element \( e_{\text{new}} \), we first identify the smallest \( i \in [0, h] \) satisfying:

\[
1 + \sum_{j=0}^{i} |S_j| \leq 2^i. \tag{5.2}
\]

We now proceed as follows:

- If \( i \) exists, we destroy \( \Upsilon_0, \Upsilon_1, \ldots, \Upsilon_i \), and move all the elements in \( S_0, S_1, \ldots, S_{i-1} \), together with \( e_{\text{new}} \), into \( S_i \) (after this, \( S_0, S_1, \ldots, S_{i-1} \) become empty). Build the structure \( \Upsilon_i \) on the current \( S_i \) from scratch.
- If \( i \) does not exist, we destroy \( \Upsilon_0, \Upsilon_1, \ldots, \Upsilon_h \), and move all the elements in \( S_0, S_1, \ldots, S_h \), together with \( e_{\text{new}} \), into \( S_{h+1} \) (after this, \( S_0, S_1, \ldots, S_h \) become empty). Build the structure \( \Upsilon_{h+1} \) on \( S_{h+1} \) from scratch. The value of \( h \) is then increased by 1.

Let us now analyze the amortized insertion cost with a charging argument. Each time \( \Upsilon_i \) (\( i \geq 0 \)) is rebuilt, we spend

\[
O(|S_i|) \cdot U(|S_i|) = O(2^i) \cdot U(2^i) \tag{5.3}
\]

cost (recall that the structure \( \Upsilon \) on \( n \) elements can be built in \( n \cdot U(n) \) time). The lemma below gives a crucial observation:

**Lemma 5.1.** Every time when \( \Upsilon_i \) is rebuilt, at least \( 1 + 2^{i-1} \) elements are added to \( S_i \) (i.e., every such element was in some \( S_j \) with \( j < i \)).

**Proof.** Set \( \lambda = i \). By the choice of \( i \), we know that, before \( S_0, \ldots, S_{\lambda-1} \) were emptied, (5.2) was violated when \( i \) was set to \( \lambda - 1 \). This means:

\[
1 + \sum_{j=0}^{\lambda-1} |S_j| \geq 1 + 2^{\lambda-1}.
\]

This proves the claim because all the elements in \( S_1, \ldots, S_{\lambda-1} \), as well as \( e_{\text{new}} \), are added to \( S_\lambda \). \( \square \)

We can therefore charge the cost of rebuilding \( \Upsilon_i \) — namely the cost shown in (5.3) — on the at least \( 2^{i-1} \) elements that are added to \( S_i \), such that each of those elements bears only

\[
\frac{O(2^i)}{2^{i-1}} \cdot U(2^i) = O(U(2^i))
\]

cost.
In other words, every time an element \( e \) moves to new \( S_i \), it bears a cost of \( O(U(2^i)) \). Note that an element never moves from \( S_i \) to an \( S_j \) with \( j < i \). Therefore, \( e \) can be charged at most \( h + 1 \) times with a total cost of

\[
O\left( \sum_{i=0}^{h} U(2^i) \right)
\]

which is the amortized cost of the insertion of \( e \). In other words, we have proved that any sequence of \( n \) insertions can be processed in

\[
O\left( n \cdot \sum_{i=0}^{h} U(2^i) \right)
\]

time.

### 5.4 Fully dynamic kd-trees with global rebuilding

Theorem 5.1 gives us a semi-dynamic version of the kd-tree. In this section, we will make the kd-tree fully dynamic, and take the chance to explain the global rebuilding technique.

#### 5.4.1 The deletion algorithm

Recall that the kd-tree on a set \( S \) of \( n \) points is a binary tree \( T \) where every point \( p \in S \) is stored at a leaf. The height of \( T \) is \( O(\log n) \).

Suppose that we need to support only deletions, but not insertions. To delete a point \( p \in S \), we carry out the following steps (assuming \( n \geq 2 \)):

1. descend a root-to-leaf path \( \Pi \) in \( T \) to find the leaf node \( z \) storing \( p \)
2. remove \( z \) from \( T \)
3. \( u \leftarrow \) the parent of \( z \); \( v \leftarrow \) the (only) child of \( u \)
4. if \( u \) is the root of \( T \) then
   5. delete \( u \), and make \( v \) the root of \( T \)
   else
   6. \( \hat{u} \leftarrow \) the parent of \( u \)
   7. delete \( u \), and make \( v \) a child of \( \hat{u} \)
8. update the MBRs of the nodes on \( \Pi \)

See Figure 5.1 for an illustration. It is easy to verify that the deletion time is \( O(\log N) \), where \( N \) is the number of points in \( S \) when the kd-tree was built (as deletions are carried out, \( n \) drops from \( N \) to 0). Note that \( T \) may appear “imbalanced” after a series of deletions.

We still answer queries using exactly the same algorithm explained in Section 4.1.2: namely, access all the nodes whose MBRs intersect with the search rectangle. Let us now discuss whether the above strategy is able to ensure \( O(\sqrt{n} + k) \) query time. The lemma gives an affirmative answer, as long as \( n \) has not dropped too much:

**Lemma 5.2.** The query time is \( O(\sqrt{n} + k) \) as long as \( n \geq N/2 \).

**Proof.** We will prove that the query time is \( O(\sqrt{N} + k) \), which is \( O(\sqrt{2n} + k) = O(\sqrt{n} + k) \). Let us recall the analysis we had in Section 4.1.2. We divided the nodes \( u \) accessed into two categories:
Figure 5.1: Deletion in a kd-tree (the deletion removes the point stored in the leaf node z)

- Type 1: the MBR of u intersects with a boundary edge of the search rectangle.
- Type 2: the MBR of u is fully contained in q.

The crux of our argument was to show that
- Claim 1: There are $O\left(\sqrt{N}\right)$ nodes of Type 1.
- Claim 2: There are $O(k)$ nodes of Type 2.

Both claims are still correct! Specifically:

- For Claim 1, first note that the claim holds right after the kd-tree was built. Now it must still hold because nodes can only \textit{disappear}, and MBRs can only \textit{shrink}.
- For Claim 2 (which was left to you as an exercise) to hold, we only need to make sure that every internal node has two child nodes. This is guaranteed by our deletion algorithm.

When $n$ is significantly less than $N$, two issues are created: (i) the query bound $O(\sqrt{n} + k)$ may no longer hold, and (ii) the height of $\mathcal{T}$ (which is $O(\log N)$) may cease to be bounded by $O(\log n)$. This is where global rebuilding comes in; this simple trick remedies both issues:

\textbf{global-rebuilding}

1. if $n = N/2$ then
2. rebuild the kd-tree on the remaining $n$ points
3. set $N = n$

What a drastic approach! But it works! Note that when we rebuild the kd-tree, $N/2$ deletions have taken place. Hence, the cost of rebuilding — which is $O((n/2) \log(n/2))$ — can be charged on those $N/2$ deletions, so that each deletion bears only

$$O \left( \frac{n \log n}{N} \right) = O( \log n)$$

cost. This increases the amortized cost of a deletion by only $O(\log n)$ because each deletion can be charged only once.

We thus have obtained a structure that consumes $O(n)$ space, answers a range reporting query in $O(\sqrt{n} + k)$ time, and supports a deletion in $O(\log n)$ amortized time.
5.4.2 Putting everything together

We can now slightly augment the logarithmic method (Section 5.3) to obtain a fully dynamic kd-tree.

As before, we divide the input set $S$ of $n$ points into disjoint subsets $S_0, S_1, ..., S_H$ (where $H = \Theta(\log n)$) satisfying $|S_i| \leq 2^i$ for every $i \in [0, H]$. Create a kd-tree $T_i$ on each $S_i$ ($i \in [0, H]$).

A query and an insertion are handled in the same way as in Sections 5.3.2 and 5.3.3, respectively. To delete a point $p$, we first locate the $S_i$ (for some $i \in [0, H]$) containing $p$. Whether $p \in S_j$ ($j \in [0, H]$) can be decided in $O(\log n)$ time using $T_j$ (we only need to descend a single root-to-leaf path in $T_j$ for this purpose); and hence, $S_i$ can be identified in $O(\log^2 n)$ time. After that, $p$ can be deleted from $T_i$ in $O(\log n)$ amortized time.

A tiny issue remains: if we have too many deletions, the value of $H$ will cease to be bounded by $O(\log n)$. This can be taken care of again by global rebuilding (think: how?). We now have obtained a data structure of $O(n)$ space that answers a range reporting query in $O(\sqrt{n} + k)$ time, and supports an update (insertion/deletion) in $O(\log^2 n)$ amortized time.
Exercises

Problem 1 (dynamic arrays). An array of size $s$ is a sequence of $s$ consecutive cells. In many operating systems, once the required space has been allocated to an array, accesses to the array are limited to that space (e.g., accessing the $(s+1)$-th cell will give a “segmentation fault” under Linux). Because of this, the size of an array is considered to be “fixed” by many people.

In this exercise, you are asked to partially remedy the above issue. Implement a data structure that stores a set $S$ of $n$ elements subject to the following requirements:

- The elements must be stored in $n$ consecutive cells.
- The space of your structure must be $O(n)$.
- An insertion to $S$ can be supported in $O(1)$ amortized time.

Problem 2. Tighten the loose end in Section 5.4.2, namely, what to do if $h$ ceases to be bounded by $O(\log n)$?

Problem 3*. Improve the amortized deletion time of our fully dynamic kd-tree to $O(\log n)$.

(Hint: currently we spend $O(\log^2 n)$ amortized time on a deletion only because we don’t know which tree contains the point to be deleted.)

Problem 4. Design a semi-dynamic data structure that stores a set of $n$ intervals in $O(n)$ space, answers a stabbing query in $O(\log^2 n + k)$ time (where $k$ is the number of intervals reported), and supports an insertion in $O(\log^2 n)$ amortized time.

Problem 5**. Design a data structure that stores a set of $n$ intervals in $O(n)$ space, answers a stabbing query in $O(\log n + k)$ time (where $k$ is the number of intervals reported), and supports a deletion in $O(\log n)$ amortized time. Your structure does not need to support insertions.

(Hint: the problem is extremely difficult if you try to delete nodes from the BST that defines the interval tree. But you don’t have to! It suffices to update only the stabbing sets, but not the BST. Show that this is okay as long as you perform global rebuilding wisely.)

Problem 6**. Let $S$ be a set of $n$ points in $\mathbb{R}^2$ that have been sorted by x-coordinate. Design an algorithm to build the priority search tree on $S$ in $O(n)$ time.

(Hint: in your undergraduate study, did you know that a max heap on $n$ real values can actually be constructed in $O(n)$ time?)

Problem 7. Design a semi-dynamic data structure that stores a set of $n$ 2D points in $O(n)$ space, answers a 3-sided range reporting query in $O(\log^2 n + k)$ time (where $k$ is the number of points reported), and supports an insertion in $O(\log n)$ amortized time.

(Hint: obviously, use the result in Problem 6.)
Lecture 6: Weight Balancing

The logarithmic method in Lecture 5 has two inherent drawbacks. First, it applies only to insertions, but not deletions. Second, it needs to search $O(\log n)$ static structures in answering a query, and thus may cause a slow-down compared to the static structure. For example, applying the technique to the interval tree (Lecture 3) results in a semi-dynamic structure that answers a stabbing query in $O(\log^2 n + k)$ time, as opposed to $O(\log n + k)$ in the static case.

We will introduce a different technique called weight balancing that allows us to remedy the above drawbacks for many structures (including the interval tree). The technique in essence is an approach to maintain a small height for a BST under updates.

6.1 BB[$\alpha$]-trees

Given a BST $T$, we denote by $|T|$ the number of nodes in $T$. Given an arbitrary node $u$ in $T$, we represent the subtree rooted at $u$ as $T_u$. Define the weight of $u$ as $|T_u|$, and its balance factor as:

$$\rho(u) = \frac{\min\{|T_1|, |T_2|\}}{|T_u|}$$

where $T_1$ (or $T_2$) is the left (or right, resp.) subtree of $u$.

Let $\alpha$ be a real-valued constant satisfying $0 < \alpha \leq 1/5$. A node $u$ in $T$ is $\alpha$-balanced if

- either $|T_u| \leq 4$
- or $\rho(u) \geq \alpha$.

In other words, either $T_u$ has very few nodes (no more than 4) or each subtree of $u$ has at least a constant fraction of the nodes in $T_u$.

$T$ is said to be a BB[$\alpha$]-tree if every node is $\alpha$-balanced (where BB stands for bounded balanced).

**Lemma 6.1.** The height of a BB[$\alpha$]-tree $T$ is $O(\log n)$, where the big-O hides a constant factor dependent on $\alpha$.

**Proof.** Let $T_1$ and $T_2$ be the left and right subtrees of $T$, respectively. By definition of BB[$\alpha$], we know that $|T_1| \leq (1 - \alpha)|T|$ and $|T_2| \leq (1 - \alpha)|T|$. In other words, each time we descend into a child, the subtree size drops by a constant factor.

Let $S$ be the set of keys in $T$; henceforth, we will consider $S$ to be a set of real values. Set $n = |S|$.

**Lemma 6.2.** If $S$ has been sorted, a BB[$\alpha$]-tree $T$ can be constructed in $O(n)$ time.
Proof. Take the median element \( e \in S \) (i.e., the \( \lceil n/2 \rceil \)-smallest in \( S \)). Create a node \( u \) to store \( e \) as the key, and make \( u \) the root of \( T \). Each subtree of \( u \) has at least \( n/2 - 1 \) nodes. If \( n \geq 4 \), the balance factor \( \rho(u) \geq \frac{n/2 - 1}{n} = 1/2 - 1/n \geq 1/4 > \alpha \). Therefore, \( u \) is \( \alpha \)-balanced.

Now, construct the left subtree of \( u \) recursively on \( \{ e' < e \mid e' \in S \} \), and the right subtree of \( u \) recursively on \( \{ e' > e \mid e' \in S \} \). The above analysis implies that every node is \( \alpha \)-balanced.

The construction time will be left as an exercise.

Corollary 6.1. After \( T \) has been constructed as in Lemma 6.2, each node with weight at least 4 has a balance factor at least 1/4.

Proof. Follows immediately from the proof of Lemma 6.2.

For each node \( u \), we store its weight along with \( u \) so that its balance factor can be calculated in constant time, once \( u \) has been identified. The space consumption of \( T \) remains \( O(n) \).

6.2 Insertion

To insert a real value \( e_{\text{new}} \) in \( S \), descend \( T \) to the leaf \( v \) whose slab (Section 2.1.2) covers \( e_{\text{new}} \). Create a node \( z \) with \( e_{\text{new}} \) as the key, and make \( z \) a child of \( v \). The cost so far is \( O(\log n) \) by Lemma 6.1.

The insertion, however, may cause some nodes to stop being \( \alpha \)-balanced. Such nodes can only appear on the path \( \Pi \) from the root to \( z \) (think: why?). Let \( u \) be the highest node that is no longer \( \alpha \)-balanced. Node \( u \), if exists, can be found in \( O(\log n) \) time.

If \( u \) does not exist, the insertion finishes. Otherwise, use Lemma 6.2 to rebuild the entire \( T_u \). The set \( S_u \) of keys in \( T_u \) can be collected from \( T_u \) in sorted order using \( O(|T_u|) \) time (depth first traversal). Therefore, \( T_u \) can be rebuilt in \( O(|T_u|) \) time.

The insertion cost therefore is bounded by \( O(\log n + |T_u|) \), which can be terribly large. However, we will show later that rebuilding a subtree occurs infrequently such that each update is amortized only \( O(\log n) \) time.

6.3 Deletion

To delete a real value \( e_{\text{old}} \) from \( S \), first find the node whose key is \( e_{\text{old}} \). For simplicity, we will consider only the case where \( e_{\text{old}} \) is the key of a leaf node \( z \) (the opposite case is left as an exercise). In that case, we simply delete \( z \) from \( T \). The cost so far is \( O(\log n) \).

The deletion may cause some nodes to violate the \( \alpha \)-balance requirement. Again, these nodes can only appear on the path \( \Pi \) from the root to \( z \). Let \( u \) be the highest node that is no longer \( \alpha \)-balanced. If \( u \) exists, rebuild \( T_u \) in the same way as in insertion.

The deletion cost is bounded by \( O(\log n + |T_u|) \). We will account for the term \( |T_u| \) with a charging argument in the next section.

6.4 Amortized analysis

Let us start with a crucial observation, which is the main reason for the usefulness of weight balancing:
Lemma 6.3. Suppose that \( T_u \) has just been reconstructed. Let \( w_u \) be the weight of \( u \) at this moment (i.e., \( w_u = |T_u| \)). Then, the next reconstruction of \( T_u \) can happen only after \( w_u/24 \) elements have been inserted or deleted in \( T_u \).

Proof. If \( w_u \leq 24 \), the lemma holds because trivially at least \( 1 \geq w_u/24 = \Omega(w_u) \) update is needed in \( T_u \) before the next reconstruction. Focus now on \( w_u \geq 24 \). By Corollary 6.1, \( \rho(u) \geq 1/4 \).

We argue that at least \( w_u/24 \) updates must have occurred in \( T_u \) before \( \rho(u) \) drops below \( \alpha \leq 1/5 \). Specifically, let \( n_1 \) be the number of nodes in the left subtree \( T_1 \) of \( u \). Hence, \( n_1 \geq w_u/4 \). Suppose that after \( x \) updates in \( T_u \), \( |T_1|/|T_u| \leq 1/5 \). We will prove that \( x \geq w_u/24 \).

After \( x \) updates, \( |T_1| \geq n_1 - x \) while \( |T_u| \leq w_u + x \). Therefore, \( |T_1|/|T_u| \geq \frac{n_1 - x}{w_u + x} \). For the ratio to be at most \( 1/5 \), we need:

\[
\frac{n_1 - x}{w_u + x} \leq 1/5 \Rightarrow 6x \geq 5n_1 - w_u \geq w_u/4 \Rightarrow x \geq w_u/24.
\]

A symmetric argument shows that at least \( w_u/24 \) updates are needed for \( |T_2|/|T_u| \leq 1/5 \) to happen, where \( |T_2| \) is the right subtree of \( u \). This completes the proof.

As a remark, the above analysis paid little efforts to minimize constants. Indeed, a more careful analysis can reduce the constant 24 considerably.

We now prove the main theorem of this lecture:

Theorem 6.1. The BB[\( \alpha \)]-tree supports any sequence of \( n \) updates (mixture of insertions and deletions) in \( O(n \log n) \) time, namely, \( O(\log n) \) amortized time per update.

Proof. It suffices to concentrate on the cost of subtree reconstruction. By Lemma 6.3, whenever a subtree \( T_u \) is rebuilt, we can charge the \( O(|T_u|) \) rebuilding cost on the \( \Omega(|T_u|) \) insertions/deletions that have taken place in \( T_u \) since the last reconstruction of \( T_u \) (we omitted some easy but subtle details here; can you spot them?). Each of those updates bears only \( O(1) \) cost.

How many times can an update be charged this way? The answer is \( O(\log n) \) because each insertion or deletion affects only \( O(\log n) \) subtrees (each subtree is rooted at a node on the update path).

6.5 Dynamization with weight balancing

The weight balancing technique can be used to dynamize all the structures in Lectures 3 and 4, except the kd-tree. Those structures have the common properties below:

- They use a BST \( T \) as the primary structure.
- Every node in \( T \) is associated with a secondary structure.

They are difficult to update because each secondary structure may be very large. Hence, when a node \( u \) of \( T \) changes its place in \( T \) (e.g., rotations in AVL-trees), we must pay a huge cost to rebuild its secondary structure, resulting in large update overhead. The weight-balancing technique remedies this issue effectively because subtree reconstructions occur very infrequently (Lemma 6.3).
To illustrate the core ideas, next we will describe an extension of the BST, which also has the two properties above, and would be exceedingly difficult to update before this lecture. Weight balancing, however, makes dynamization almost trivial. The same ideas apply to other structures as well.

6.5.1 Dynamic arrays
Let \( S \) be a set of \( n \) elements. A dynamic array \( A \) on \( S \) is an array satisfying the following:

- \( A \) has size \( O(n) \).
- The elements of \( S \) are stored in the first \( n \) positions of \( A \) (ordering is not important).
- \( A \) supports an insertion on \( S \) in \( O(1) \) amortized time.

The design of a dynamic array was an exercise of Lecture 5.

6.5.2 A BST augmented with dynamic arrays
Let \( T \) be a BST created on a set \( S \) of \( n \) real values. For each node \( u \) in \( T \), denote by \( T_u \) the subtree rooted at \( u \), and by \( S_u \) the set of keys in \( T_u \). Create a dynamic array \( A_u \) on \( S_u \), and make \( A_u \) the secondary structure of \( u \).

Let us try to support insertions while maintaining the \( O(\log n) \) height of \( T \). It is easy to update \( T \) itself, e.g., using rotations as in the AVL-tree. However, when rotation changes the position of a node \( u \) in \( T \), \( S_u \) can change significantly, and hence, so can \( A_u \). The size \( |S_u| \) can be very large (in the worst case, \( \Omega(n) \)), because of which rebuilding \( A_u \) will incur terrible update cost!

6.5.3 Replacing the BST with a BB\([\alpha]\)-tree
Now, redefine \( T \) to be a BB\([\alpha]\)-tree on \( S \). The meanings of \( T_u \), \( S_u \), and \( A_u \) are the same as before. An insertion can now be supported easily in \( O(\log^2 n) \) amortized time.

Given a real value \( e_{\text{new}} \), we first create a new leaf \( z \) in \( T \) with \( e_{\text{new}} \) as the key. This takes \( O(\log n) \) time by following a root-to-\( z \) path \( \Pi \). For every node \( u \) on \( \Pi \), \( e_{\text{new}} \) is inserted into \( A_u \) in \( O(1) \) amortized time. The cost so far is \( O(\log n) \) amortized.

The insertion procedure of the BB\([\alpha]\)-tree (Section 6.2) may need to reconstruct the subtree \( T_u \) of a node \( u \) on \( \Pi \). When this happens, we simply reconstruct all the secondary arrays in \( T_u \) in \( O(|S_u| \log |S_u|) = O(|S_u| \log n) \) time. By Lemma 6.3, \( \Omega(|S_u|) \) updates must have taken place in \( T_u \) since the last reconstruction of \( T_u \). Each of those updates is therefore charged only \( O(\log n) \) time for the reconstruction of \( A_u \).

An update can be charged only \( O(\log n) \) times (\( e_{\text{new}} \) is in \( O(\log n) \) subtrees), and hence, has amortized cost \( O(\log^2 n) \).

6.6 Remarks
Our definition is one of the many ways to describe the BB\([\alpha]\] tree. See [10] for the original proposition.

The BB\([\alpha]\]-tree (our version in Section 6.1) can actually be updated in \( O(\log n) \) worst-case time. Whenever a node \( u \) stops being \( \alpha \)-balanced, it can be fixed in \( O(1) \) time by either a single-rotation
or a double-rotation, just like the AVL-tree. Even better, after a fix on a node $u$, the node $u$ can violate $\alpha$-balance only after $\Omega(w_u)$ updates have taken place in $T_u$. The details can also be found in [3].

Finally, it is worth mentioning that the structure in Section 6.5.2 is useful for independent query sampling; see [9].
Exercises

Problem 1. Prove the construction time in Lemma 6.2.

Problem 2. Complete the deletion algorithm for the case where $e_{old}$ is the key of an internal node. (Hint: the “standard” strategy for BSTs suffices.)

Problem 3 (dynamic arrays with deletions). Let $S$ be a set of $n$ elements. Design a data structure with the properties below:

- the structure stores an array $A$ of size $O(n)$.
- The elements of $S$ are stored in the first $n$ positions of $A$ (ordering is not important).
- An insertion/deletion on $S$ can be supported in $O(\log n)$ amortized time.

Problem 4. Explain how to support an insertion/deletion on the structure of Section 6.5.3 in $O(\log^2 n)$ amortized time.

Problem 5. Explain how to support an insertion/deletion on the interval tree (Section 3.1) in $O(\log^2 n)$ amortized time, where $n$ is the number of intervals. Your structure must still be able to answer a stabbing query in $O(\log n + k)$ time, where $k$ is the number of intervals reported.

Problem 6. Explain how to support an insertion/deletion on the priority search tree (Section 4.3) in $O(\log^2 n)$ amortized time, where $n$ is the number of points. Your structure must still be able to answer a 3-sided range query in $O(\log n + k)$ time, where $k$ is the number of points reported.

Problem 7*. Improve the update time in the previous problem to $O(\log n)$.

Problem 8. Explain how support an insertion/deletion on the range tree (Section 4.4) in $O(\log^3 n)$ amortized time, where $n$ is the number of points. Your structure must still be able to answer a 4-sided range query in $O(\log^2 n + k)$ time, where $k$ is the number of points reported.
Lecture 7: Partial Persistence

A dynamic data structure is usually *ephemeral* because, once updated, its previous version is lost. For example, consider \( n \) insertions into an initially empty BST. At the end, we have a BST with \( n \) nodes (the final version). However, \( n - 1 \) other versions had been created in history (one after each of the first \( n - 1 \) insertions); all those versions have been lost.

Wouldn’t it be nice if we could retain all versions? One naive approach to do so is to store a separate copy of each past version, which requires \( O(n^2) \) space. Amazingly, we will learn a powerful technique called *partial persistence* that allows us to achieve the purpose in just \( O(n) \) space (which is clearly optimal).

The technique in fact is applicable to *any* pointer-machine structure (Section 4.6), as long as each node in the structure has a constant in-degree (for the BST, the in-degree is 1). This includes most of the structures you already know: the linked list, the priority queue, all the structures in Lectures 3 and 4, and so on (but not dynamic arrays).

The implication of this technique goes beyond just retaining history. It can be used to solve difficult problems using surprisingly primitive structures. One example is the 3-sided range query that we dealt with using the priority search tree in Section 4.3. As we will see in an exercise, that problem can be settled by simply making the BST partially persistent.

### 7.1 The potential method

This is a generic method for *amortized* analysis we will apply later. As it may not have been covered at the undergraduate level, we include an introduction here.

Consider \( n \) operations on a data structure, the \( i \)-th (\( 1 \leq i \leq n \)) of which has cost \( C_i \). We want to argue that \( \sum_{i=1}^{n} C_i \) is small, or equivalently, the amortized cost per operation is small.

Define a function \( \Phi \) which maps the current structure to a real value. Let \( T_0 \) be the structure before all operations, and \( T_i \) (\( 1 \leq i \leq n \)) be the structure after operation \( i \). Define for each \( i \in [1, n] \):

\[
\Delta_i = \Phi(T_i) - \Phi(T_{i-1}).
\]  

(7.1)

We can now claim:

**Lemma 7.1.** If \( \Phi(T_n) \geq \Phi(T_0) \), the amortized cost of operation \( i \) is at most \( C_i + \Delta_i \).

*Proof.* It suffices to prove \( \sum_{i=1}^{n} C_i \leq \sum_{i=1}^{n} (C_i + \Delta_i) \). This is obvious because

\[
\sum_{i=1}^{n} \Delta_i = \Phi(T_n) - \Phi(T_0) \geq 0.
\]

\( \square \)
Why do we want to claim that the amortized cost is $c_i + \Delta_i$, instead of $c_i$? This is because $\Delta_i$ can be negative! Indeed, a successful argument under the potential method must be able to assign a negative $\Delta_i$ to offset every large $C_i$.

It is worth mentioning that $\Phi$ is called a potential function.

7.2 Partially persistent BST

Starting with an empty BST $T_0$, we will process a sequence of $n$ updates (mixture of insertions and deletions). The $i$-th $(1 \leq i \leq n)$ update is said to happen at time $i$. Denote by $T_i$ the BST after the update, which is said to be of version $i$. Our goal is to retain the BSTs of all versions.

We will refer to the BST of the latest version as the live BST, and denote it as $T$. In other words, after $i$ updates, the live BST is $T = T_i$.

Denote by $A$ the update algorithm of the BST, which can be any implementation of the BST, e.g., the AVL-tree, the red-black tree, the BB[$\alpha$]-tree, etc.

7.2.1 The first attempt

Our first idea is to enforce the principle that whenever $A$ needs to change a node $u$, make a copy of $u$, and apply the changes on the new copy.

Example. Consider the update sequence that inserts 8, 4, 12, and 14. Figure 7.1(a) shows the live BST $T = T_1$, which contains a single node. The node information has the format “[i]k”, to indicate that the node is created at time $i$ with key $k$.

To perform the second insertion, we create a node “[2] 4”, and need to make it the left child of “[1] 8”. Following the aforementioned principle, “[1] 8” is not altered; instead, we copy it to “[2] 8”, and set “[2] 4” the left child of “[2] 8”. As shown in Figure 7.1(b), both BSTs $T_1$ and $T_2$ are explicitly stored.

To insert 12, we create a node “[3] 12”, which ought to be the right child of “[2] 8”. Following the principle, we copy “[2] 8” to “[3] 8”, and set “[3] 12” as the right child of “[3] 8”. The structure now is shown in Figure 7.1(c). Note that the left child of “[3] 8” is still “[2] 4” (“[2] 4” is not copied because it is not modified by this update). Observe that Figure 7.1(c) implicitly stores 3 BSTs $T_1$, $T_2$, $T_3$.

Figure 7.1(d) presents the final structure after inserting 14, which encodes BSTs $T_1$, ..., $T_4$.  

We will call the above method naive copying. Since each update on the live BST accesses
$O(n \log n)$ nodes, naive copying can create $O(n \log n)$ nodes per update in the persistent structure. The overall space consumption is therefore $O(n \log n)$.

Any BST in the past can be found and searched efficiently. For any $i \in [1, n]$, the root of $T_i$ can be identified in $O(\log n)$ time (by creating a separate BST on the root versions). Then, the search can proceed within $T_i$ in same manner as a normal BST is searched.

The drawback of naive copying is that it sometimes copies a node that is not modified by $A$. In Figure 7.1(d), for example, the only node that “really” needs to be modified is “[3] 12” but the method duplicates all its ancestors. This motivates our next improvement.

### 7.2.2 An improved method

The new idea is to introduce a modification field in each node $u$. When $A$ needs to change the pointer of $u$, the change is recorded in the field. Only when the field has no more room will we resort to node copying. It turns out that a field of constant size suffices to reduce the space to $O(n)$.

Each node now takes the form $\{(i, k, ptr_1, ptr_2), mod\}$ where

- the first component $((i, k, ptr_1, ptr_2)$ indicates that the node is created at version $i$ with key $k$ and pointers $ptr_1$ and $ptr_2$ (which may be NULL);
- the second component $mod$ is the modification field, which is empty when the node is created, and can log exactly one pointer change.

**Example.** We will first insert 8, 4, 12, 14, 2, and then delete 2, 14. Figure 7.2(a) shows the structure after the first insertion. Here, the $ptr_1$ and $ptr_2$ of node I are both NULL. The empty space on the right of the vertical bar indicates an empty $mod$.

To insert 4, we create node II, and make it the left child of node I. This means redirecting the left pointer of node I to node II at time 2. This pointer change is described in the $mod$ of node I, as shown in Figure 7.2(b). Observe how the current structure encodes both $T_1$ and $T_2$.

The insertion of 12 creates node III, which should be the right child of node I. As the $mod$ of node I is already full, we cannot log the pointer change inside node I. We thus resort to node copying. As shown in Figure 7.2(c), this spawns node IV which stores “[3] 8”, and has $ptr_1$ and $ptr_2$ referencing nodes II and III, respectively. The current structures encodes $T_1, T_2,$ and $T_3$.

14 and 2 are then inserted in the same manner, as illustrated by Figures 7.2(d) and (e), respectively.

The next operation deletes 2. Accordingly, we should reset the pointer of node II to NULL (which removes node VI from the live tree). Since node II’s $mod$ is full, we copy it to node VII. This, in turn, requires changing the left pointer of node IV, as is recorded in its $mod$. The current structure in Figure 7.2(f) encodes $T_1, ..., T_6$.

Finally, the deletion of 14 requires nullifying the right pointer of node III. As Node III’s $mod$ is full, it is copied to node VIII, which further triggers node IV to be copied to node IX. Figure 7.2(g) gives the final structure which encodes $T_1, ..., T_7$.

In general, $A$ can change the live BST with two operations:

- C-operation: creating a new node $u$. This happens only in insertion, and the node created stores the key being inserted. Accordingly, we also create a node in the persistent structure.
Figure 7.2: Illustration of the improved method on the update sequence of inserting 8, 4, 12, 14, 2 followed by deleting 2 and 14.

- P-operation: updating a pointer of a node $u$. We do so in the persistent structure as follows:

  \[\text{ptr-update}(u)\]
  \begin{enumerate}
  \item if the mod of $u$ is empty then
  \item record the pointer change in mod
  \item return
  \item else /* mod full */
  \item copy $u$ to node $v$
  \item $\hat{u} \leftarrow$ the parent of $u$ in the live BST
  \item call \texttt{ptr-update}(\hat{u}) to add a pointer from $\hat{u}$ to $v$
  \end{enumerate}

  Note that Line 7 may recursively invoke \texttt{ptr-update} and thereby induce multiple node copies.

  The time to build a persistent BST is clearly $O(n \log n)$ (the proof is left to you as an exercise). As in Section 7.2.1, we can identify the root of any $\mathcal{T}_i$ ($1 \leq i \leq n$) in $O(\log n)$ time, after which $\mathcal{T}_i$ can then be navigated as a normal BST.
We will analyze the space consumption in the next subsection.

### 7.2.3 Space

Denote by \( m_i \) \((1 \leq i \leq n)\) the number of C/P-operations that \( A \) performs on the live tree in processing the \( i \)-th update. We will prove:

**Lemma 7.2.** The algorithm in Section 7.2.2 creates \( O(\sum_{i=1}^{n} m_i) \) nodes in the persistent tree.

The lemma immediately implies:

**Theorem 7.1.** Given a sequence of \( n \) updates on an initially empty BST, we can build a persistent BST of \( O(n) \) space in \( O(n \log n) \) time.

**Proof.** The red-black tree performs at most one C-operation and \( O(1) \) P-operations in each insertion/deletion. \( \square \)

**Proof of Lemma 7.2.** Set

\[
M = \sum_{i=1}^{n} m_i
\]

namely, \( M \) is the total number of C/P-operations performed by \( A \). These operations happen in succession, and hence, can be ordered as operation 1, 2, ..., \( M \).

Let \( C_j \) \((1 \leq j \leq M)\) be the number of nodes (in the persistent tree) created by the \( j \)-th operation. We will prove \( \sum_{j=1}^{M} C_j = O(M) \), or equivalently, each operation creates \( O(1) \) nodes amortized.

Denote by \( S_j \) \((1 \leq j \leq M)\) the set of nodes in the live tree after the \( j \)-th operation. Define specially \( S_0 \) the empty set. Define a potential function \( \Phi \) that maps \( S_j \) to a real value; specifically, \( \Phi(S_j) \) equals the number of nodes in \( S_j \) whose information fields are non-empty. Clearly, \( \Phi(S_M) \geq \Phi(S_0) = 0 \).

By Lemma 7.1, after amortization operation \( j \) creates at most

\[
C_j + \Phi(S_j) - \Phi(S_{j-1})
\]

(7.2)

nodes. The remainder of the proof will show that the above is precisely 1 for every \( j \), which will complete the proof of Lemma 7.2.

If operation \( j \) is a C-operation, it creates a node with an empty \( mod \) and finishes. Hence, \( C_j = 1 \), and \( S_j = S_{j-1} \). Therefore, (7.2) equals 1.

Now, consider that operation \( j \) is a P-operation. Every new node is created by node copying (Line 5 of ptr-update). However, every time this happens, we lose a node with non-empty \( mod \), and create a node with empty \( mod \). At the end of the P-operation, we fill in the \( mod \) of one node, thus converting it from a node with empty \( mod \) to one with non-empty \( mod \). Therefore, \( \Phi(S_j) - \Phi(S_{j-1}) \) equals precisely \(-C_j + 1\) such that (7.1) also equals 1.

### 7.3 General pointer-machine structures

The following result generalizes Theorem 7.1:
Theorem 7.2 ([8]). Consider any pointer-machine structure defined in Section 4.6 where every node has a constant in-degree. Suppose that $A$ is an algorithm used to process a sequence of $n$ updates (mixture of insertions and deletions) with amortized update cost $U(n)$. Let $m_i$ be the number of nodes created/modified by $A$ in processing the $i$-th update ($1 \leq i \leq n$). Then, we can create a persistent structure that records all the historical versions in $O(n \cdot U(n))$ time. The structure consumes $O(\sum_{i=1}^{n} m_i)$ space. The root of every version can be identified in $O(\log n)$ time.

For example, if the structure is the linked list, then $U(n) = O(1)$ and $m_i = O(1)$. Therefore, we can construct a persistent linked list of $O(n)$ space in $O(n)$ time. The head node of the linked list of every past version can be identified in $O(\log n)$ time.

The theorem can be established using the modification-logging approach in Section 7.2.2, except that the modification field should be made sufficiently large (but still have a constant size). The proof makes an interesting, but not compulsory, exercise.
Exercises

Problem 1. Prove the construction time in Theorem 7.1.

Problem 2. Let $S$ be a set of $n$ horizontal rays in $\mathbb{R}^2$, each having the form $[x, \infty) \times y$. Explain how to store $S$ in a persistent BST of $O(n)$ space such that, given any vertical segment $q = x \times [y_1, y_2]$, we can report all the rays in $S$ intersecting $q$ using $O(\log n + k)$ time, where $k$ is the number of rays reported.

Problem 3. Let $P$ be a set of $n$ points in $\mathbb{R}^2$. Explain how to store $P$ in a persistent BST of $O(n^2)$ space such that any 3-sided range query of the form $(-\infty, x] \times [y_1, y_2]$ can be answered in $O(\log n + k)$ time, where $k$ is the number of points reported. (Hint: Problem 2.)

Problem 4. Let $P$ be a set of $n$ points in $\mathbb{R}^2$. Given an axis-parallel rectangle $q$, a range count query reports the number of points in $P$ that are covered by $Q$. Design a structure that stores $P$ in $O(n \log n)$ space that can answer a range count query in $O(\log n)$ time. 

Remark: this improves an exercise in Lecture 4. (Hint: persistent count BST.)

Problem 5. Prove Theorem 7.2 for the linked list.

Remark: the persistent linked list is one way to store all the past versions of a document that is being edited (regard a document as a sequence of characters.)

Problem 6* (point location). A polygonal subdivision of $\mathbb{R}^2$ is a set of non-overlapping convex polygons whose union is $\mathbb{R}^2$. The following shows an example (for clarity, the boundary of $\mathbb{R}^2$ is represented as a rectangle).

![Diagram of a polygonal subdivision](image)

Given a point $q$ in $\mathbb{R}^2$, a point location query reports the polygon that contains $q$ (if $q$ falls on the boundary of more than one polygon, any such polygon can be reported).

Let $n$ be the number of segments in the subdivision. Design a structure of $O(n)$ space that can answer any point location query in $O(\log n)$ time. (Hint: persistent BST.)

Problem 7**. Prove Theorem 7.2.
Lecture 8: Dynamic Perfect Hashing

In the dictionary search problem, we want to store a set $S$ of $n$ integers in a data structure to answer the following queries efficiently: given an integer $q$, report whether $q \in S$ (the output is boolean: yes or no). At the undergraduate level, we have learned that the problem can be tackled with hashing. Specifically, we can store $S$ in a hash table of $O(n)$ space which answers a query in $O(1)$ expected time.

In practice, we may not be satisfied with $O(1)$ expected query cost because it implies that the actual search time can still be large occasionally. Ideally, we would like to build a perfect hash table that guarantees $O(1)$ query cost in the worst case.

This lecture will introduce a technique called cuckoo hashing which can be used to maintain a perfect hash table of $O(n)$ size with $O(1)$ amortized expected time per update (what this means will be defined formally later). We will, however, establish only a weaker bound of $O(\log n)$; as a benefit in return, this illustrates nicely how data structures can arise from graph theory.

8.1 Two random graph results

Let $U, V$ each be a set of $c \cdot n \geq 2$ vertices, for some integers $c > 0, n > 0$. We generate a random bipartite graph $G$ by repeating the gen-edge operation $n$ times:

- gen-edge
  1. pick a vertex $u \in U$ uniformly at random
  2. pick a vertex $v \in V$ uniformly at random
  3. connect $u, v$ with an edge (if the edge does not exist)

Lemma 8.1. When $c \geq 2e^2$, it holds with probability at least $7/8$ that $G$ contains no cycles.

Proof. Consider any integer $\ell \in [4, n]$. We will prove an upper bound on the probability that $G$ has a cycle of length $\ell$. Let us start with:

$$\Pr[\text{a cycle of length } \ell] \leq \sum_{\Sigma} \Pr[\Sigma \text{ induces a cycle in } G] \quad (8.1)$$

where the summation is over all $\Sigma \subseteq U \cup V$ with $|\Sigma| = \ell$.

Next, we will analyze $\Pr[\Sigma \text{ induces a cycle in } G]$ for an arbitrary $\Sigma = \{u_1, u_2, \ldots, u_\ell\}$.

$$\Pr[\Sigma \text{ induces a cycle in } G] \leq \sum_{I} \Pr[\text{all gen-edge operations indexed by } I \text{ create an edge on } \Sigma] \quad (8.2)$$

where the summation is over all $I \subseteq \{1, 2, \ldots, n\}$ with $|I| = \ell$; and the set of operations indexed by $I$ is the set of operations $\{\text{operation } i \mid i \in I\}$. 

52
**gen-edge** creates an edge on Σ with probability at most \((\frac{ℓ}{cn})^2\) because both \(u\) and \(v\) it chooses must fall in Σ. It follows that

\[
\Pr[\text{all gen-edge operations indexed by } I \text{ create an edge on } Σ] \leq \left(\frac{𝓁}{cn}\right)^{2ℓ}
\]

and hence by (8.2)

\[
\Pr[Σ \text{ induces a cycle in } G] \leq \left(\frac{n}{ℓ}\right) \cdot \left(\frac{𝓁}{cn}\right)^{2ℓ}
\]

with which (8.1) gives

\[
\Pr[\text{a cycle of length } ℓ] \leq \left(\frac{cn}{ℓ}\right) \cdot \left(\frac{n}{ℓ}\right) \cdot \left(\frac{𝓁}{cn}\right)^{2ℓ} \leq \left(\frac{e \cdot cn}{ℓ}\right)^{ℓ} \cdot \left(\frac{e \cdot n}{ℓ}\right)^{ℓ} \cdot \left(\frac{ℓ}{cn}\right)^{2ℓ} = \left(\frac{e^2}{c}\right)^{ℓ} \leq (1/2)^ℓ.
\]

We now prove the lemma with

\[
\Pr[\text{G has a cycle}] \leq \sum_{ℓ=4}^{n} \Pr[\text{a cycle of length } ℓ] \leq \sum_{ℓ=4}^{n} (1/2)^ℓ < 1/8.
\]

An almost identical argument establishes:

**Lemma 8.2.** When \(c \geq 2e^3\), it holds with probability at least \(1 - \frac{c}{n^x}\) that \(G\) has no simple path longer than \(4 \log_2 n\) edges (a path is simple if it passes no vertex twice).

The proof is left as an exercise.

### 8.2 Cuckoo hashing

#### 8.2.1 Amortized expected update cost

As defined in Section 5.1, a structure has *amortized* update cost \(T_{udp}\) if it supports any sequence of \(n \geq 1\) updates in \(n \cdot T_{udp}\) total time. Specifically, if \(X_i\) (\(1 \leq i \leq n\)) is the cost of the \(i\)-th update, it must hold that \(\sum_{i=1}^{n} X_i \leq n \cdot T_{udp}\).

Now consider the structure to be randomized such that each \(X_i\) is a random variable. In this case, the structure has *amortized expected* update cost \(T_{udp}\) if it guarantees \(E[\sum_{i=1}^{n} X_i] \leq n \cdot T_{udp}\), which means \(\sum_{i=1}^{n} E[X_i] \leq n \cdot T_{udp}\).
8.2.2 Hash functions

Denote by \( \mathbb{D} \) the domain from which the elements of \( S \) are drawn. A hash function \( h \) maps \( \mathbb{D} \) to a set of integers \( \{1, 2, ..., N\} \) for some \( N \geq 1 \). The output \( h(e) \) is the hash value of \( e \in \mathbb{D} \). We will assume uniform hashing, which means:

- for any element \( e \in \mathbb{D} \), \( \Pr[h(e) = i] = 1/N \) for any \( i \in [1, N] \);
- the above holds regardless of the hash values of the other elements in \( \mathbb{D} \).

8.2.3 The hash table, query, and deletion

We maintain two arrays \( A, B \) each of size \( N = O(n) \) where the concrete value of \( N \) will be chosen later. There are two hash functions \( g \) and \( h \), both mapping \( \mathbb{D} \) to \( \{1, ..., N\} \). We enforce:

**Invariant:** Each element \( e \in S \) is stored at either \( A[g(e)] \) or \( B[h(e)] \).

This makes querying and deletions very simple:

- **Query:** Given an element \( q \), report yes if \( A[g(e)] \) or \( B[h(e)] = q \); otherwise, report no.
- **Deletion:** To delete an element \( e \in S \), erase \( A[g(e)] \) or \( B[h(e)] \) whichever equals \( e \).

Clearly, both operations finish in \( O(1) \) worst-case time.

8.2.4 Insertion

To insert an element \( e_{\text{new}} \), if \( A[g(e_{\text{new}})] \) is empty, we store \( e_{\text{new}} \) at \( A[g(e_{\text{new}})] \) and finish. Otherwise, if \( B[h(e_{\text{new}})] \) is empty, we store \( e_{\text{new}} \) at \( B[h(e_{\text{new}})] \) and finish.

If both \( A[g(e_{\text{new}})] \) and \( B[g(e_{\text{new}})] \) are occupied, we launch a *bumping process* which can be intuitively understood as follows. Remember every element \( e \in S \) has two “nests”: \( A[g(e)] \) and \( B[h(e)] \). If \( e \) is evicted from one nest, we are obliged to store it in the other. With this mentality, let us put \( e_{\text{new}} \) at \( A[g(e_{\text{new}})] \) *anyway*, thus forcing the original element \( e \) there to be evicted. Thus, \( e \) must go into its other nest in \( B \), thereby evicting another element there. The process then goes on until all the elements have been placed properly. There is a chance that this may not be possible, in which case we declare failure.

Formally, we begin the bumping process by calling \texttt{bump}(e_{\text{new}}):
bump(e)
1. turn = g; cnt = 0
2. while cnt ≤ 4 log₂ n do
3.    cnt++
4.    if turn = g then
5.       if A[g(e)] empty then
6.          place e at A[g(e)]; return success
else
7.       swap e and A[g(e)]; turn = h
else
8.       /* turn = h */
9.       if B[h(e)] empty then
10.      place e at B[h(e)]; return success
else
11.     swap e and B[h(e)]; turn = g
12. return failure

Note that functions g and h are used in a round-robin fashion.

Example. Set N = 4. Currently, S contains a single element 10. Suppose that g(10) = 2, h(10) = 3, and that 10 is stored in A[g(10)] = A[2].

Consider 16 as the next insertion, for which we assume g(16) = 2 and h(16) = 4. In other words, A[g(16)] = A[2] is occupied, but B[h(16)] = B[4] is empty. We thus store 16 at B[4], and finish. Now A = (−, 10, −, −) and B = (−, −, −, 16).

The next insertion is 35, for which we assume g(35) = 2 and h(35) = 4. As both A[g(35)] and B[h(35)] are both occupied, a bumping process is launched. The process stores 35 at A[g(35)] = A[2], and forces the original element 10 there to relinquish its place. For element 10, we find B[h(10)] = B[3] empty, and thus store 10 there. The insertion finishes with A = (−, 35, −, −) and B = (−, −, 10, 16).

We receive one more insertion 29 with g(29) = 2 and h(29) = 4. As A[2] and B[4] are occupied, the bumping process removes 35 from A[g(35)] = A[2], and stores 29 there instead. Currently, A = (−, 29, −, −). As h(35) = 4, the process replaces 16 with 35 at B[4], turning B to (−, −, 10, 35). The process then puts 16 at A[g(16)] = A[2], and remove element 29, after which A = (−, 16, −, −). The process continues in this manner and eventually declares failure.

If the bumping process fails, we simply rebuild the whole structure:

rebuild
1. choose another two hash functions g and h
2. insert the elements of S one by one, and stop if failure declared
3. if Line 2 fails then
4.    repeat from Line 1

Note that Line 2 can fail because it may invoke the bumping process.

8.2.5 Checkpoint rebuild

We ensure the following constraint on the array size N:

\[2e^3 \cdot n \leq N \leq 8e^3 \cdot n.\] (8.3)
This can be achieved with global rebuilding (Section 5.4). Specifically, before processing the first update (which must be an insertion), we place a checkpoint, and set $N$ is set to $4e^3$. In general, after $N/(8e^3)$ updates since the previous checkpoint, another checkpoint occurs, such that we reconstruct the structure by calling rebuild (Section 8.2.4) with array size $N = 4e^3 \cdot n$ (where $n$ the current size of $S$).

**Lemma 8.3.** Equation (8.3) holds at all times.

**Proof.** Let $n_{chk}$ be the size of $S$ at the last checkpoint. There can be at most $n_{chk} + N/(8e^3) = 1.5n_{chk}$ distinct elements in $S$ till the next checkpoint. Hence, at all times we have $2e^3 \cdot n \leq 2e^3 \cdot 1.5n_{chk} < N$. On the other hand, there are at least $n_{chk} - N/(8e^3) = 0.5n_{chk}$ elements in $S$ till the next checkpoint. Hence, at all times we have $n \geq 0.5n_{chk} = N/(8e^3)$.

### 8.3 Analysis

This section will prove:

**Theorem 8.1.** Fix any sequence of $n$ updates (mixture of insertions and deletions). The above algorithm maintains a perfect hash table under the updates in $O(n \log n)$ total expected time.

The core of the proof is to establish:

**Lemma 8.4.** Consider any checkpoint. Let $N$ be the array size set at the checkpoint. The total cost of the following tasks is $O(N \log N)$ expected:

- rebuilding the structure at the checkpoint;
- performing the next $N/(8e^3)$ updates (i.e., until the next checkpoint).

The lemma implies Theorem 8.1. To see this, first notice that The lemma implies $\Omega(N)$ updates between the previous checkpoint and the current one. Therefore, we can charge the $O(N \log N)$ cost on those updates such that each is amortized only $O(\log N)$ cost expected.

#### 8.3.1 Proof of Lemma 8.4

We will prove only the first bullet, because essentially the same argument also proves the second bullet, which is left as an exercise.

We start by establishing a vital connection between cuckoo hashing and random graphs. Set $U = V = \{1, 2, ..., N\}$. For each an element $e \in S$, create an edge between vertex $g(e) \in U$ and vertex $h(e) \in V$. Let $G$ be the bipartite graph obtained. As $g(e)$ (or $h(e)$) chooses each vertex in $U$ (or $V$, resp.) with the same probability, $G$ is precisely a random graph obtained in Section 8.1.

**Corollary 8.1.** With probability at least $1/2$, $G$ has both the properties below:

- $G$ has no cycles.
- $G$ has no simple path of longer than $4 \log_2 n$ edges.
Proof. Consider first $n \leq 16$. In this case, $G$ obviously cannot have a simple path of length $4 \log_2 n = 16$ because such a path needs 17 vertices. By Lemma 8.1, $G$ has the first property with probability at least $7/8$.

Consider now $n > 16$. Lemma 8.1 shows that the first property can be violated with probability at most $1/8$. Since (8.3) always holds, Lemma 8.2 indicates that the second property can be violated with probability at most $(2e^3)/n^2 \leq (2e^3)/16^2 < 1/4$. Hence, the probability for at least one property to be violated is no more than $1/8 + 1/4 < 1/2$. \hfill \Box

**Lemma 8.5.** Line 2 of **rebuild** (Section 8.2.4) takes $O(n \log n)$ time.

**Proof.** Line 2 performs $n$ insertions at Line 2. Each insertion takes $O(1)$ time if no bumping process is required. Otherwise, it takes $O(\log n)$ time before declaring success or failure. \hfill \Box

**Lemma 8.6.** If $G$ has both properties in Corollary 8.1, Line 2 of **rebuild** successfully builds the entire structure.

**Proof.** We will prove that, with the two properties, the bumping process will never fail. This will establish the lemma.

When the bumping process evicts an element $e$ from one nest to the other — say from $A[g(e)]$ to $B[h(e)]$ — we cross an edge in $G$ from vertex $g(e) \in U$ to $h(e) \in V$. Therefore, if the process fails, we must have traveled on a path $\Pi$ of more than $4 \log_2 n$ edges.

Wait! Since $G$ has no cycles, $\Pi$ cannot pass two identical vertices. But then $\Pi$ must be a simple path, which is not possible either. This creates a contradiction. \hfill \Box

We can now put together Corollary 8.1, Lemmas 8.5 and 8.6 to prove that **rebuild** finishes in $O(n \log n)$ expected time. Let $X$ be the number of times that Line 2 is executed. By Corollary 8.1 and Lemma 8.6, every time Line 2 is executed, it fails with probability at most $1/2$, which indicates that $\Pr[X = t] \leq (1/2)^{t-1}$. Lemma 8.5 implies that the total cost of **rebuild** is $O(X \cdot n \log n)$. Therefore, the expected cost is

$$\sum_{t=1}^{\infty} O(t \cdot n \log n) \cdot \Pr[X = t] = \sum_{t=1}^{\infty} O(t \cdot n \log n) \cdot \left(\frac{1}{2}\right)^{t-1} = O(n \log n).$$

This completes the proof of the first bullet of Lemma 8.4.

### 8.4 Remarks

Our discussion of cuckoo hashing emphasized on its relationships to random graphs. The analysis presented, however, is loose in at least two ways. First, as mentioned, cuckoo hashing actually achieves $O(1)$ amortized expected time per update. Second, our hash table can have a size up to $8e^3 n$ (see (8.3)), which can be reduced considerably. We refer the interested students to [12].

Our assumption of uniform hashing can also be relaxed, although this takes more efforts. As shown in [12], $O(\log n)$-wise independent hashing (i.e., intuitively this means that any $O(\log n)$ hash values are guaranteed to be independent; our assumption is essentially $n$-wise independence) is good enough, but the analysis would have to deviate significantly from the two lemmas in Section 8.1. It is worth noting that there exist $O(\log n)$-wise independent hash functions that can be evaluated in constant expected time; see [13].
Exercises

Problem 1. Prove Lemma 8.2.

Problem 2. Prove the second bullet of Lemma 8.4 assuming all those \( N/(8e^3) \) updates are insertions.

(Hint: pretend all those insertions were given at the checkpoint and include them in the argument for proving the first bullet.)

Problem 3. Prove the second bullet of Lemma 8.4 in general (i.e., allowing deletions).

Problem 4 (a uniform hashing function requires lots of space to represent). Let \( \mathbb{D} \) be the set of integers from 1 to \( D \) where \( D \geq 1 \) is an integer.

(a) How many different functions are there mapping \( \mathbb{D} \) to \( \{1, 2, ..., N\} \) where \( N \geq 1 \) is an integer?

(b) Prove: at least \( D \log_2 N \) bits are required to represent all the above functions, regardless of how the functions are encoding in binary form.

(c)∗ Prove: if any uniform-hashing function from \( \mathbb{D} \) to \( \{1, 2, ..., N\} \) requires \( D \log_2 N \) bits to represent. (Hint: such a hash function must be a random variable. What are the possible values of this random variable?)

Remark: this means that uniform hashing may not be a fair assumption for practical applications.

The next two exercises would help you gain intuition as to why cuckoo hashing guarantees \( O(1) \) expected amortized update time.

Problem 5. Consider a checkpoint rebuild where \( N = 4e^3n \) and \( n = |S| \). Recall that the rebuild algorithm (Section 8.2.4) inserts the elements of \( S \) one by one. Let \( e \in S \) be the last element inserted. Prove: when \( e \) is inserted, \( A[g(e)] \) is occupied with probability at most \( \frac{1}{4e^3} \).

(Hint: for any \( e' \neq e \), \( \Pr[g(e) = g(e')] = 1/N. \))

Remark: this means the insertion of \( e \) requires no bumping process with probability at least \( 1 - \frac{1}{4e^3} > 98\% \).

Problem 6. Same settings as in Problem 5. Suppose that the insertion of \( e \) launches the bumping process. Recall that the process evicts a sequences of elements; let the sequence be \( e_1, e_2, ..., e_\ell \).

(a) Prove: \( \Pr[\ell > 1] \leq \frac{1}{4e^3} \).

(b) Assume that \( e, e_1, e_2 \) are distinct. Prove: \( \Pr[\ell > 2] \leq \left( \frac{1}{4e^3} \right)^2 \).

(c) Assume that \( e, e_1, ..., e_t \) are distinct. Prove: \( \Pr[\ell > t] \leq \left( \frac{1}{4e^3} \right)^t \).

(Hint: if you can solve (a), you can solve the rest. For (a), think of something similar to Problem 5.)

58
Bibliography


