Lecture Notes On Data Structures

Andreas Avgoustis

Greece, November 2025

Preface

The content of this work is a careful selection and detailed presentation of the most important data structures, based on lectures given at the Department of Informatics of the University of Saarland, Germany, and the Department of Computer Engineering and Informatics of the University of Patras.

The main bibliographic source used was the book by K. Mehlhorn "Data Structures and Algorithms I: Sorting and Searching" (Springer Verlag, 1984) and newer original papers.

Significant introductory and clarifying texts, as well as the basic division of the material, partially coincide with the work of Mehlhorn and Tsakalidis (Mehlhorn K. and A.K. Tsakalidis "Chapter 6: Data Structures", Handbook of Theoretical Computer Science, Elsevier Science Publishers, 1990), which presents a general overview of data structures.

I believe that this work presents significant ideas of practical theory, and I hope it will be a helpful aid to every student and scientist interested in structured and non-trivial programming.

I also hope that through the study of this work, the conviction will spread more and more in our country that the development of quality software requires a full understanding of the theory of data structures.

Contents

Chapter 1

Introduction

1.1 Models of Computation

Before referring to algorithm analysis, it is necessary to study the computational models on which our algorithms are executed. The models we will examine are the **Random Access Machine (RAM)** and the **Pointer Machine (PM)**.

A computing machine according to the Random Access Machine (RAM) model has the following characteristics:

- It has 4 Registers, 1 Accumulator, Index Registers, and an infinite number of memory locations numbered from 0.
- Memory access is possible via address calculation.
- Instructions are one-address; that is, each instruction refers to only one memory location.

The Pointer Machine (PM) model shares characteristics with RAM but does not allow access via address calculation; access is only possible via pointers.

For the following discussion, we assume a **Unit Cost RAM** model, where each operation takes constant time. If data length exceeds one memory word, we use the **Logarithmic Cost RAM** model, where operations cost proportional to the length of the data ($\approx \log n$).

1.2 Algorithm Complexity

We are primarily interested in the total number of steps an algorithm executes for a given input of size n. Let us consider the `FINDMAX` algorithm.

The complexity $T(n)$ for FINDMAX is $T(n) = 4(n-1) + 2M + 3$ operations, where M is the number of times lines 4-5 are executed.

1.3 Analysis Cases

We typically distinguish three cases of complexity:

1. **Worst Case Complexity:** The maximum number of steps for any input of size n. For FINDMAX, the worst case is a sorted sequence, where $T(n) = 5n - 2$.
2. **Average Case Complexity:** We calculate the average cost over all possible inputs I_i with probability p_i: $\overline{T}(n) = \sum_{i=1}^{n!}(p_i C_i(n))$. For FINDMAX, $\overline{T}(n) \approx 3n + 2\ln n - 2$.

Algorithm 1 FINDMAX(A, n)

```
1: i ← 1; m ← A[1];
2: j ← 2;
3: while j ≤ n do
4:     if A[j] > m then
5:         m ← A[j];
6:         i ← j;                          ▷ m is s.t. A[i] ≥ A[k], 1 ≤ k ≤ j
7:     end if
8:     j ← j + 1;
9: end while
10: return (m, i);
```

3. **Amortized Complexity:** We bound the cost of a sequence of operations rather than a single operation. The amortized complexity of an operation O_i is $\frac{\sum C(O_i)}{m}$.

1.4 Asymptotic Notations

We use asymptotic notations to describe the growth rate of functions:

- **Big-O (O):** $g(n) = O(f(n))$ if $g(n) \leq c \cdot f(n)$ for $n \geq n_0$ (Upper bound) [cite: 52-54].
- **Omega (Ω):** $g(n) = \Omega(f(n))$ if $g(n) \geq c \cdot f(n)$ for $n \geq n_0$ (Lower bound).
- **Theta (Θ):** $g(n) = \Theta(f(n))$ if $c_1 f(n) \leq g(n) \leq c_2 f(n)$ (Tight bound).

1.5 Recurrence Relations

Recurrence relations are used to analyze recursive algorithms.

1.5.1 Linear Recurrences

First Order: $a_n = x_n a_{n-1} + y_n$.

- If $y_n = 0$: $a_n = a_0 \prod_{i=1}^{n} x_i$ [cite: 74-75].
- If $x_n = 1$: $a_n = a_0 + \sum_{i=1}^{n} y_i$.

1.5.2 Recursion Tree

Sometimes, visualizing a recurrence relation via a 'recursion tree' helps in understanding what happens when the relation is repeated. Consider the recurrence $T_n = 3T_{n/3} + bn$. As shown in Figure 1.1, for an instance of size n, it executes bn steps plus the steps for three subproblems of size $n/3$.

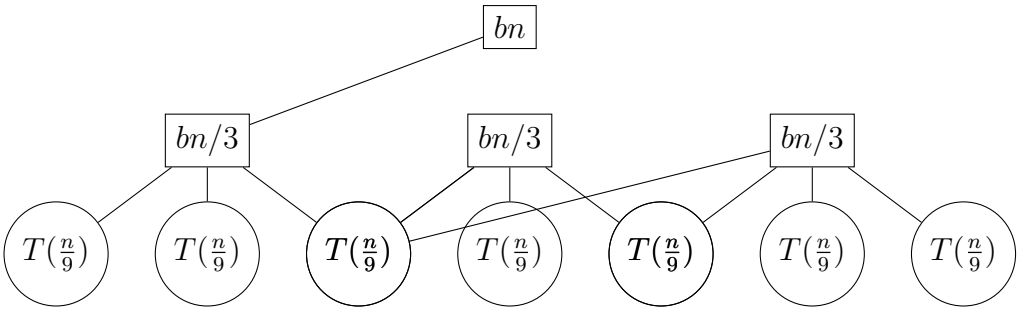

Figure 1.1: Figure 1.1: Recursion Tree for $T(n) = 3T(n/3) + bn$. The total work at depth k is $(3^k)(b \cdot n/3^k) = bn$.

1.5.3 The Master Theorem

For recurrences of the form $T_n = aT_{n/b} + f(n)$ with $a \geq 1, b > 1$:

1. If $f(n) = O(n^{\log_b a - \epsilon})$, then $T_n = \Theta(n^{\log_b a})$.
2. If $f(n) = \Theta(n^{\log_b a})$, then $T_n = \Theta(n^{\log_b a} \log n)$.
3. If $f(n) = \Omega(n^{\log_b a + \epsilon})$ and regularity holds, then $T_n = \Theta(f(n))$.

1.6 Structured Data Types

1.6.1 Basic Types

Basic types include Integers, Reals, Characters, and Pointers.

1.6.2 Arrays

An array consists of a fixed number of elements of the same type. Each element can be accessed in $O(1)$ time. Arrays are typically stored in contiguous memory locations.

int A[1..5]

Figure 1.2: Figure 1.3: Representation of an Array in Memory.

1.6.3 Lists

Lists are used to represent sequences of elements. Their main advantage over arrays is their dynamic nature. Each element contains content and a pointer to the next element.

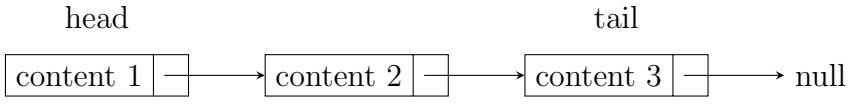

Figure 1.3: Figure 1.5: A Singly Linked List.

Operations include:

- `first()`, `last()`: $O(1)$
- `insert_after(ptr, data)`: $O(1)$
- `remove(ptr)`: $O(1)$
- `find(data)`: $O(n)$

Lists can also be **Doubly Linked** (Figure 1.8), where each element points to both the next and the previous element, or **Circular**.

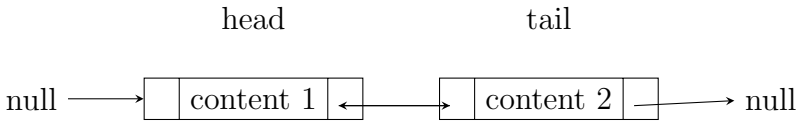

Figure 1.4: Figure 1.8: A Doubly Linked List.

1.6.4 Stacks and Queues

- **Stack (LIFO):** Insertions and deletions occur at one end (top). Operations: `push`, `pop` [cite: 218-225].
- **Queue (FIFO):** Insertions at the end, deletions at the front. Operations: `push`, `pop` [cite: 246-254].

1.6.5 Trees

A tree is a connected, acyclic, undirected graph. **Definitions**:

- **Root:** The distinguished node at the top.
- **Leaf:** A node with no children.
- **Leaf-Oriented Tree:** Data is stored only at the leaves.

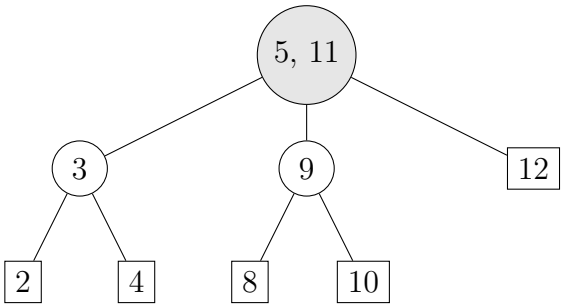

Figure 1.5: Figure 1.16: Leaf-Oriented Search Tree for S={2,4,8,10,12}.

Traversals:

- **Preorder:** Visit Root → Children.
- **Postorder:** Visit Children → Root.

- **Symmetric (Inorder):** Left Child → Root → Right Child.

Search Trees Operations:

- `Search(x)`: Find if x exists.
- `Insert(x)`: Add x if not present.
- `Delete(x)`: Remove x.

If data is inserted in sorted order, a tree may degenerate into a list, increasing complexity to $O(n)$.

1.6.6 Abstract vs. Concrete Data Types

An **Abstract Data Type (ADT)** is defined by a set of values and operations (e.g., Dictionary). A **Concrete Data Structure** is the implementation (e.g., Hash Table) [cite: 351-353].

Chapter 2

Sorting

The problem of sorting is located in finding an algorithmic procedure that brings order to a set, placing its elements in ascending order .

More formally, the problem can be stated as follows: Given a set S consisting of n elements $S =< s_1, s_2, ..., s_n >$ for which a linear order is defined (i.e., there is a function that takes two elements s_i, s_j and decides if $s_i < s_j, s_i = s_j$ or $s_i > s_j$), the goal is to produce a permutation (rearrangement) of the elements, $< s'_1, s'_2, ..., s'_n >$, such that:

$$s'_1 \leq s'_2 \leq ... \leq s'_n \tag{2.1}$$

A primary categorization of sorting algorithms is based on the operations performed on the elements of S:

- **Comparison-based algorithms:** These algorithms produce the output using only comparisons between elements. Examples include Bubble Sort, Insertion Sort, Heap Sort, Merge Sort, and Quicksort. We will prove later that any comparison-based algorithm requires $\Omega(n \log n)$ comparisons to sort a set of size n.

- **Non-comparison-based algorithms:** These algorithms use specific information about the input to achieve better times (e.g., that all numbers are integers). Examples include Counting Sort, Radix Sort, and Bucket Sort[cite: 360].

We assume the input is given as an array $S[1..n]$. If the space required by an algorithm, besides the input array, is $O(1)$, the algorithm is called **in-place**[cite: 361].

2.1 Sorting in Main Memory

2.1.1 Bubble Sort

The algorithm is based on the following idea: The entire array S is scanned from the beginning, and every time a pair of elements with $S[i] > S[i+1]$ is encountered, the elements are swapped. This method results in the largest element of the array moving to position $S[n]$ during the first pass. Then the scan is repeated for the section $S[1..n-1]$, and so on[cite: 368].

The maximum number of comparisons executed by the algorithm is given by:

$$T(n) = (n-1) + (n-2) + ... + 1 = \frac{n(n-1)}{2} = O(n^2) \tag{2.2}$$

Algorithm 2 BUBBLESORT(S, n)

```
1: up ← n;
2: while up > 1 do
3:     j ← 0;                                   ▷ j keeps the number of swaps
4:     for i = 1 to up − 1 do
5:         if S[i] > S[i + 1] then
6:             Swap S[i], S[i + 1];
7:             j ← j + 1;
8:         end if
9:     end for
10:    if j = 0 then return ;                   ▷ No swap occurred, S is sorted
11:    end if
12:    up ← up − 1;
13: end while
```

2.1.2 Insertion Sort

We assume that in the first j passes, $j \geq 2$, we have already sorted the subarray $S[1..j]$. In the next step, we read element $S[j+1]$ and scan the sorted section $S[1..j]$ to find the appropriate position to insert it[cite: 389].

Algorithm 3 INSERTIONSORT(S, n)

```
1: for j ← 2 to n do
2:     key ← S[j];
3:     i ← j − 1;                  ▷ Insert S[j] into the sorted sequence S[1..j − 1]
4:     while i > 0 AND key < S[i] do
5:         S[i + 1] ← S[i];                          ▷ Shift elements right
6:         i ← i − 1;
7:     end while
8:     S[i + 1] ← key;
9: end for
```

In the worst case (reverse sorted input), the complexity is:

$$T(n) = \sum_{i=2}^{n} i = \frac{n(n+1)}{2} - 1 = O(n^2) \tag{2.3}$$

2.1.3 Heap Sort

Heap Sort is a sorting method with optimal asymptotic performance. It uses a data structure called a **balanced binary heap**.

Definition: A balanced binary heap is a binary tree with the following properties:

1. Each node v contains a value from S.
2. The **Heap Property** holds: $value(parent(v)) \geq value(v)$. This implies the largest element is at the root.
3. The tree is balanced: all leaves are at depth k or $k+1$, and leaves at depth $k+1$ are left-aligned.

A heap can be represented by an array S, where for a node at index k, its parent is at $\lfloor k/2 \rfloor$ and its children are at $2k$ and $2k+1$.

Heap Sort consists of two phases[cite: 444]:

1. **Construction Phase:** Convert the input array $S[1..n]$ into a heap.
2. **Selection Phase:** Repeatedly remove the maximum element (root), place it at the end of the array, and restore the heap property for the remaining elements.

Algorithm 4 HEAPSORT(S, n)

```
1: l ← ⌊n/2⌋ + 1; r ← n;
2: while r ≥ 2 do
3:     if l > 1 then
4:         l ← l − 1;                                  ▷ Construction Phase
5:     else
6:         Swap S[1], S[r];                            ▷ Selection Phase
7:         r ← r − 1;
8:     end if
9:     j ← l; s ← S[j];                                ▷ Sift-down procedure starts
10:    while 2j ≤ r do
11:        k ← 2j;
12:        if k < r AND S[k] < S[k + 1] then k ← k + 1;
13:        end if
14:        if s < S[k] then
15:            S[j] ← S[k]; j ← k;
16:        else
17:            break;
18:        end if
19:    end while
20:    S[j] ← s;
21: end while
```

Complexity: The Construction Phase takes $O(n)$. The Selection Phase takes $O(n \log n)$ because removing the maximum element takes $O(\log n)$ and is repeated n times. Thus, the total worst-case complexity is $O(n \log n)$.

2.1.4 Merge Sort

Merge Sort is another optimal algorithm based on the Divide and Conquer technique:

1. **Divide:** Split the input into two halves.
2. **Conquer:** Recursively sort the two halves.
3. **Combine:** Merge the two sorted subsequences into one sorted sequence.

The recurrence relation for the number of comparisons is $M(n) = 2M(n/2) + n - 1$, which gives $M(n) = O(n \log n)$.

Algorithm 5 MERGESORT(S, n)

Split S into subsequences S_1 and S_2 of size $\lfloor n/2 \rfloor$ and $\lceil n/2 \rceil$.
if $|S_1| > 1$ **then** MERGESORT(S_1);
end if
if $|S_2| > 1$ **then** MERGESORT(S_2);
end if
Merge sorted S_1 and S_2 into Z;
return Z;

2.1.5 Quicksort

Quicksort is also a Divide and Conquer algorithm. It partitions the array based on a **pivot** element (e.g., $S[1]$). All elements smaller than the pivot are moved to its left, and all larger elements to its right. This process is applied recursively.

Algorithm 6 QUICKSORT(l, r)

$i \leftarrow l$; $k \leftarrow r + 1$; $pivot \leftarrow S[l]$;
while $i < k$ **do**
    repeat
        $i \leftarrow i + 1$;
    until $S[i] \geq pivot$
    repeat
        $k \leftarrow k - 1$;
    until $S[k] \leq pivot$
    if $i < k$ **then**
        Swap $S[i], S[k]$;
    end if
end while
Swap $S[l], S[k]$;
if $l < k - 1$ **then** QUICKSORT$(l, k - 1)$;
end if
if $k + 1 < r$ **then** QUICKSORT$(k + 1, r)$;
end if

Complexity:

- **Worst Case:** $O(n^2)$. This happens when the partition is extremely unbalanced (e.g., pivot is always the minimum).
- **Average Case:** $O(n \log n)$. This assumes all permutations of input are equally likely.

2.2 Lower Bound for Comparison-Based Sorting

Using the decision tree model, it can be proven that any algorithm that sorts by comparing elements requires $\Omega(n \log n)$ comparisons in the worst case.

2.3 Sorting Integers

If we know more about the input (e.g., elements are integers in a specific range), we can sort faster than $O(n \log n)$.

2.3.1 Counting Sort

Counting Sort assumes elements are integers in $[1..k]$. It counts the occurrences of each number to determine its position in the sorted array. The complexity is $O(n + k)$.

2.3.2 Radix Sort

Radix Sort sorts integers by processing them digit by digit (from least significant to most significant), using a stable sort (like a variant of Counting Sort) for each digit position. For d digits and base k, the complexity is $O(d(n + k))$.

Chapter 3

Finding the i-th Largest Element

The problem we are called to address in this chapter is as follows: Given a sequence $S = s_1, s_2, ..., s_n$ with distinct elements, and an integer i, $1 \leq i \leq n$. We wish to find the i-th smallest element, that is, an element x_j such that there are $i-1$ elements x_l with $x_l < x_j$ and $n-i$ elements x_t with $x_t > x_j$.

Definition 3.1. The $\lceil n/2 \rceil$-th largest element is called the **median**.

A naive solution to the problem would be to sort S, in which case the i-th element results from direct access to the i-th position of the sorted array. This solution has a complexity dictated by the sorting algorithms, requiring $\Theta(n \log n)$ time in the worst and average cases.

Below, we will describe two solutions to the problem: the **Find** algorithm, which runs in $O(n^2)$ worst-case time but $O(n)$ on average, and the **Select** algorithm, which is linear ($O(n)$) in both the average and worst cases.

3.1 The Find Algorithm

The Find algorithm is based on a core idea similar to that of Quicksort. The initial sequence is partitioned into two subsequences based on a random element, say s. The first subsequence, let's call it M_1, contains elements smaller than s, and the second, M_2, contains elements larger than s. The size of M_1 is critical for the algorithm's continuation. If $|M_1| = i-1$, then the i-th element is s. If $|M_1| < i-1$, the element is in M_2, so we recursively apply the algorithm to M_2. If $|M_1| > i-1$, the element is in M_1, so we recurse there. This is another application of the "Divide and Conquer" technique.

Algorithm 7 FIND(M, i)

1: $s \leftarrow$ A random element from M;
2: $M_1 \leftarrow \{m \in M : m < s\}$; ▷ Elements smaller than the pivot
3: $M_2 \leftarrow \{m \in M : m > s\}$; ▷ Elements larger than the pivot
4: **case** $|M_1|$ **of**
5: $\quad < i-1$: FIND$(M_2, i - |M_1| - 1)$;
6: $\quad = i-1$: **return** s; ▷ s is the i-th element
7: $\quad > i-1$: FIND(M_1, i);
8: **end case**

When the sequence is represented as an array, the partition can be performed similarly to Quicksort. The time required for partitioning is $O(|M|)$. Thus, the total time is:

$$T_{Find} = O(|M|) + \text{Time for recursive calls} \tag{3.1}$$

The worst case occurs when the partition is extremely unbalanced (e.g., $|M_1| = 0$ in every step), leading to $T_{Find} = O(n^2)$.

3.1.1 Average Case Analysis

Assuming distinct elements and that every permutation is equally likely, it can be proven that the Find algorithm requires linear time on average. Let $T(n)$ be the maximum average time required for any i. It can be shown by induction that $T(n) \leq 4cn$, where c is the cost per element for the partitioning step. Therefore: **Theorem 3.2.** The Find algorithm requires linear time in the average case.

3.2 The Linear Algorithm Select

The Select algorithm guarantees linear time even in the worst case by ensuring a good partition split. The idea is to choose the pivot based on a better sample. We split M into groups of 5 elements, find the median of each group, and then recursively find the median of these medians (let's call it $\overline{m}$). This $\overline{m}$ is guaranteed to be a good pivot.

Algorithm 8 SELECT(M, i)

```
1: n ← |M|;
2: if n ≤ 100 then
3:     Sort M and return the i-th element directly;
4: else
5:     Split M into subsets M_1, M_2, ..., M_⌈n/5⌉ of 5 elements each;
6:     Sort each M_j and find its median m_j;
7:     m̄ ← SELECT({m_1, ..., m_⌈n/5⌉}, ⌈n/10⌉);        ▷ Find median of medians
8:     M_1 ← {m ∈ M : m < m̄};
9:     M_2 ← {m ∈ M : m > m̄};
10:    if i ≤ |M_1| then
11:        SELECT(M_1, i);
12:    else if i = |M_1| + 1 then return m̄;
13:    else
14:        SELECT(M_2, i − |M_1| − 1);
15:    end if
16: end if
```

The process of splitting into groups of 5 and finding the median of medians is illustrated in Figure 3.1.

3.2.1 Complexity Analysis

It can be proven that the maximum size of the recursive subproblem (M_1 or M_2) is at most $\frac{8n}{11}$ (roughly $0.7n$). The recurrence relation for the time complexity is:

$$T(n) \leq T\left(\frac{21n}{100}\right) + T\left(\frac{8n}{11}\right) + bn \tag{3.2}$$

where $T(21n/100)$ is the cost to find the median of medians, and bn is the cost for partitioning. Solving this recurrence proves that:

$$T(n) = O(n) \tag{3.3}$$

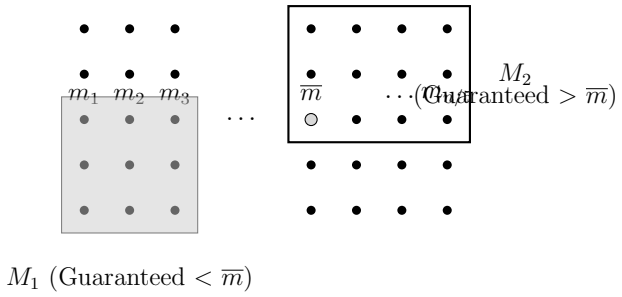

Figure 3.1: Figure 3.1: Division into fives. The grey box represents elements guaranteed to be smaller than $\overline{m}$, and the outlined box represents elements guaranteed to be larger than $\overline{m}$.

Theorem 3.5. The Select algorithm runs in linear time with respect to the size of the input.

Chapter 4

The Dictionary Problem

Let U be a universe containing all elements that may ever be of interest, and a set $S \subseteq U$. Each element $x \in S$ also carries auxiliary information $inf(x)$. For example, if S consists of the names of an airline's customers, then $inf(x)$, $x \in S$, might contain the flight number or other details.

The **Dictionary** is an Abstract Data Type (ADT) that supports the following operations:

- `Access(x)`: Returns `true` along with the field $inf(x)$ if $x \in S$, and `false` otherwise.
- `Insert(x)`: Replaces S with $S \cup \{x\}$ and associates the information $inf(x)$ with x.
- `Delete(x)`: Replaces S with $S - \{x\}$.

The operations `Insert(x)` and `Delete(x)` are destructive, meaning they destroy the version of the set S to which they are applied, yielding a new set S'. Structures operating this way are called **ephemeral**. Structures that support these operations non-destructively are called **persistent**.

Structures supporting only the `Access` operation are called **static**, whereas those supporting the full repertoire of operations are called **dynamic**. A further distinction is made based on the information used:

- **Comparison-based:** Use only comparisons between elements (e.g., Binary Search, Search Trees).
- **Representation-based:** Use the representation of elements (e.g., Tries, Hashing).

Finally, structures are categorized by space usage:

- **Succinct (Implicit) Structures:** Occupy exactly $n + O(1)$ space.
- **Explicit Structures:** Occupy $O(n)$ space (e.g., using pointers).

4.1 Succinct Data Structures - Introduction

Succinct data structures occupy exactly $n + O(1)$ space and are usually implemented using the RAM model and arrays. Here we refer to static succinct structures. Let $S \subseteq U$ with $|S| = n$ be stored in increasing order in an array $S[1..n]$. The basic program for the search operation is:

The method for choosing `next` determines the time complexity:

Algorithm 9 ACCESS(x)

```
1: left ← 1; right ← n;
2: next ← a number in [left...right];
3: while (x ≠ S[next] AND left < right) do
4:     if x ≤ S[next] then
5:         right ← next − 1;
6:     else
7:         left ← next + 1;
8:     end if
9:     next ← a number in [left...right];
10: end while
11: if x = S[next] then
12:     return Success;
13: else
14:     return Failure;
15: end if
```

- $next \leftarrow left + 1$: **Linear Search** ($O(n)$ worst case).
- $next \leftarrow \lfloor \frac{right+left}{2} \rfloor$: **Binary Search**.
- $next \leftarrow \lfloor \frac{x-S[left]}{S[right]-S[left]}(right - left) \rfloor + left$: **Interpolation Search**.

4.2 Binary Search

In binary search, at each step, we check the middle of the interval. The interval is halved in each iteration. Starting with size n, after h steps the interval has size 1. Thus, $n(1/2)^h = 1 \Rightarrow h = \log_2 n$.

Binary search requires $O(\log n)$ time in the worst case.

This process can be visualized as a binary tree (Figure 4.1).

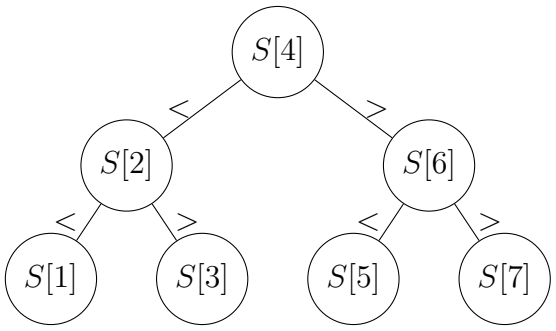

Figure 4.1: Binary Search Tree representation for an array of size 7.

4.3 Interpolation Search

Interpolation search mimics how a human searches a dictionary. We estimate the position of x based on the values at the ends of the interval. The next position is calculated as:

$$next \leftarrow \lfloor \frac{x - S[left]}{S[right] - S[left]} (right - left) \rfloor + left \tag{4.1}$$

Worst-case time is $O(n)$ (e.g., when values increase exponentially), but the average time is $O(\log \log n)$ for uniform distributions.

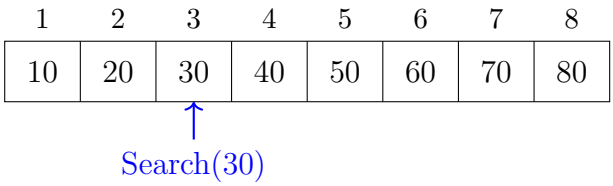

Figure 4.2: Concept of Interpolation Search: estimating the position.

4.4 Binary Interpolation Search

Binary Interpolation Search (BIS) combines the ideas to achieve $O(\sqrt{n})$ worst-case time and $O(\log \log n)$ average time.

Algorithm 10 ACCESS(x) - Binary Interpolation Search

1: $left \leftarrow 1; right \leftarrow n;$
2: $size \leftarrow right - left + 1;$
3: $next \leftarrow \lceil size \cdot \frac{x - S[left]}{S[right] - S[left]} \rceil + 1;$
4: **while** $(x \neq S[next])$ **do**
5: $i \leftarrow 0$; $size \leftarrow right - left + 1;$
6: **if** $size \leq 3$ **then** Linear Search;
7: **end if**
8: **if** $x \geq S[next]$ **then**
9: **while** $x > S[next + i\sqrt{size} - 1]$ **do** $i \leftarrow i + 1;$
10: **end while**
11: $right \leftarrow next + i\sqrt{size};$
12: $left \leftarrow next + (i - 1)\sqrt{size};$
13: **else**
14: **while** $x < S[next - i\sqrt{size} + 1]$ **do** $i \leftarrow i + 1;$
15: **end while**
16: $right \leftarrow next - (i - 1)\sqrt{size};$
17: $left \leftarrow next - i\sqrt{size};$
18: **end if**
19: Recalculate *next* via interpolation on new bounds;
20: **end while**
21: **return** Success/Failure;

Essentially, after the interpolation step, we search linearly with jumps of size $\sqrt{size}$ to locate the sub-interval.

Theorem 4.2. The average time of Binary Interpolation Search is $O(\log \log n)$. **Theorem 4.3.** The worst-case time is $O(\sqrt{n})$.

4.4.1 Improving Worst-Case Time

The worst-case time can be improved to $O(\log n)$ without affecting the average case. Instead of increasing i linearly ($i \leftarrow i + 1$) inside the loop, we increase it exponentially ($i \leftarrow 2 * i$). This allows us to cover larger distances and quickly find the sub-interval.

4.5 Interpolation Search for Unknown Non-Uniform Distributions

Standard interpolation search works well for uniform distributions. For unknown distributions, [Willard] proposed two methods:

1. **Alternate:** In each iteration, alternate between calculating *next* via Binary Search and Interpolation Search.

2. **Retrieve:** Select *next* via a combined formula involving $\sqrt{n}$ and parameters to adapt to the distribution.

Theorem 4.4. When elements are chosen from $[0, 1]$ with a smooth (regular) distribution, the average time for Alternate is $O(\sqrt{\log n})$ and for Retrieve is $O(\log \log n)$.

Chapter 5

Explicit Data Structures

In this chapter, we will examine structures that occupy additional space for the representation of dictionary elements and consequently belong to the category of **explicit** data structures. We will examine **balanced search trees**. These are trees where the leaves are at a distance of the same order of magnitude from the root.

To maintain balance, specific balancing criteria are defined, based on which we distinguish balanced trees into two categories:

- **Height Balanced Trees:** The balancing criterion for each node is the height of its subtrees. Examples: AVL trees, B-trees, (a,b)-trees, Red-Black trees.
- **Weight Balanced Trees:** The balancing criterion for each node is the weight of its subtrees (i.e., the number of leaves stored in the subtrees). Examples: BB[α]-trees.

5.1 Balanced Trees

5.1.1 The AVL Tree

The AVL tree (named after Adelson-Velsky and Landis) is a binary balanced tree where, for every node, the heights of its subtrees differ by at most one. Let v be an internal node and $L(v), R(v)$ the subtrees rooted at the left and right children of v, respectively. We define the **height balance** $hb(v)$ as:

$$hb(v) = \text{Height}(R(v)) - \text{Height}(L(v)) \tag{5.1}$$

Permissible values for $hb(v)$ are $+1, 0, -1$. A node with $hb(v) = 0$ is called balanced.

Rotations

Insertions and deletions can violate the AVL property. To restore balance, we perform rotations.

- **Single Rotation:** Used when the imbalance is in the "outer" subtrees (Left-Left or Right-Right).
- **Double Rotation:** Used when the imbalance is in the "inner" subtrees (Left-Right or Right-Left).

Theorem 5.4. The operations `Access`, `Insert`, and `Delete` in an AVL tree with n leaves take $O(\log n)$ time in the worst case.

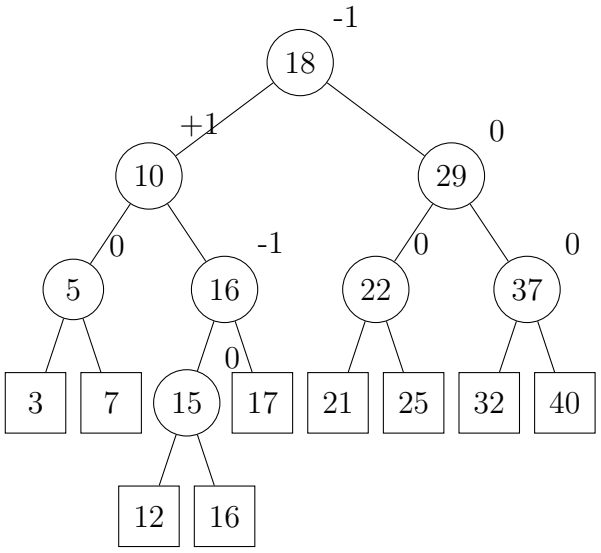

Figure 5.1: Figure 5.1: An AVL tree example. The balance factor is shown next to each internal node.

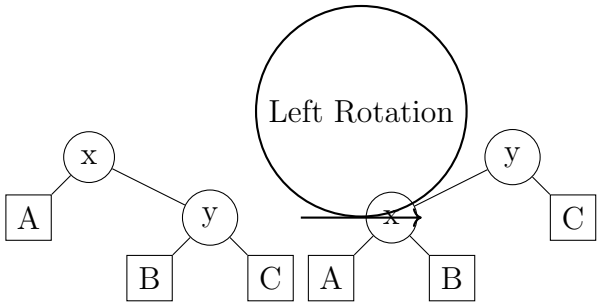

Figure 5.2: Single Left Rotation.

5.1.2 The (a, b)-Tree

(a, b)-trees are a class of balanced search trees where each node has between a and b children.
Definition 5.5. Let integers a, b with $a \geq 2$ and $2a - 1 \leq b$. A tree T is an (a, b)-tree if:

1. All leaves are at the same depth.
2. Every node v (except the root) has degree $d(v)$ such that $a \leq d(v) \leq b$.
3. The root has degree $2 \leq d(root) \leq b$.

If $b = 2a - 1$, the tree is called a **B-tree**.

Operations:

- **Insertion:** If a node overflows (children $> b$), it is **Split** into two nodes.
- **Deletion:** If a node underflows (children $< a$), it performs a **Share** (borrowing from a sibling) or **Fuse** (merging with a sibling) operation.

Theorem 5.11. An (a, b)-tree supports `Access`, `Insert`, and `Delete` in $O(\log n)$ worst-case time.

5.1.3 Red-Black Trees

A **Red-Black Tree** is a binary search tree where each node is colored either red or black, satisfying the following properties:

1. The root is black.
2. All leaves are black.
3. If a node is red, then both its children are black.
4. Every path from the root to a leaf contains the same number of black nodes (Black Height).

Red-black trees are essentially a binary representation of (2, 4)-trees.

5.2 Amortized Analysis Techniques

Amortized analysis guarantees the average performance of each operation in the worst-case sequence of operations.

- **Aggregate Method:** We compute the total cost of a sequence of n operations and divide by n.
- **Banker's Method:** We assign "credits" to specific parts of the data structure. Cheap operations pay extra credits, which are saved to pay for expensive operations later.
- **Physicist's (Potential) Method:** We define a potential function Φ that maps the state of the data structure to a real number. The amortized cost is the actual cost plus the change in potential ($\hat{c}_i = c_i + \Phi(D_i) - \Phi(D_{i-1})$).

5.3 Self-Organizing Data Structures

Self-organizing structures adjust their layout based on access patterns to improve performance for frequently accessed elements.

5.3.1 Splay Trees

A **Splay Tree** is a self-adjusting binary search tree. The core operation is **Splaying**: whenever a node x is accessed, it is moved to the root via a sequence of rotations. **Splay Steps:**

- **Zig Case:** If x's parent $p(x)$ is the root. Perform a single rotation.
- **Zig-Zig Case:** If x and $p(x)$ are both left or both right children. Rotate $p(x)$ with $p(p(x))$, then x with $p(x)$.
- **Zig-Zag Case:** If x is a right child and $p(x)$ is a left child (or vice-versa). Rotate x with $p(x)$, then x with $p(p(x))$ (Double rotation).

Chapter 6

Hybrid Data Structures

The structures presented in this chapter combine the use of pointers and arrays, which is why they are called **hybrid**. The main representatives of this class of data structures are Digital Search Trees (TRIES) and the Interpolation Search Tree (IST).

6.1 Digital Search Trees (TRIES)

Instead of relying on comparing the values of elements $x_i \in S$ to represent them in a structure, we can alternatively use the representation of these elements as a sequence of numbers or characters. For example, consider a dictionary that has tabs for the letters of the alphabet. Given a word, we access the section for its first letter. Then, within that section, we could imagine a sub-index for the second letter, and so on.

Example: Let $S = \{121, 102, 211, 120, 210, 212\}$ and the alphabet be $K = \{0, 1, 2\}$. Following the logic where for each digit (from left to right) we have a directory, we get the structure called a **TRIE**.

Definition 6.1. Let U be a universe whose elements are strings of length l over an alphabet K with $|K| = k$ (i.e., $U = K^l$). A set $S \subseteq U$ is represented as a k-ary tree containing all prefixes of the elements of S.

A common implementation for the TRIE is for each internal node to be an array of size k of pointers. Each position in the array corresponds to a letter of the alphabet.

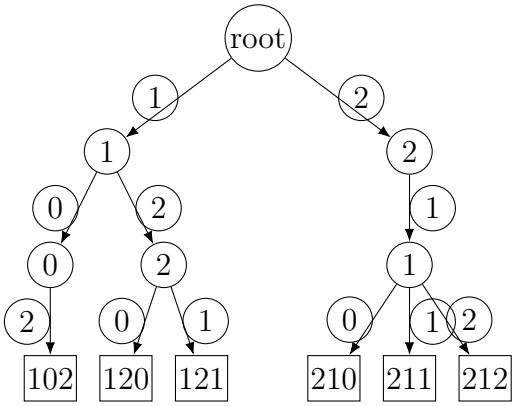

Figure 6.1: Figure 6.1: A Trie for S={102, 120, 121, 210, 211, 212} over alphabet {0,1,2}.

6.1.1 Operations

The operations `Access`, `Insert`, and `Delete` are implemented by traversing the tree based on the digits of x.

Algorithm 11 INSERT(x)

```
1: u ← root of TRIE;
2: for i = 0 to l − 1 do
3:     if x[i]-th child of u is NULL then
4:         Create the x[i]-th child;
5:     end if
6:     u ← x[i]-th child of u;
7: end for
8: content(u) ← x;
```

All operations require $O(l)$ time, where l is the number of digits. If the universe size is N, then $l = \log_k N$. Thus, the time complexity is $O(\log_k N)$. The space complexity is $O(n \cdot k)$ in the worst case (when no elements share common prefixes).

6.1.2 Compressed Tries

To avoid space wastage when there are long chains of nodes with a single child, we can use **Compressed Tries** (or Compact Tries). Chains of single-child nodes are replaced by a single edge labeled with a string of characters or a skip count.

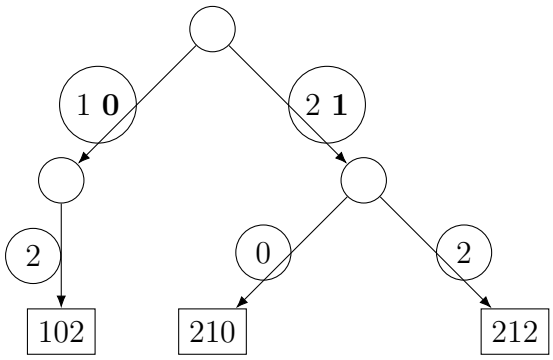

Figure 6.2: Figure 6.4: A Compressed Trie. Chains are collapsed.

6.2 Suffix Trees

A Suffix Tree for a string X is a data structure representing all suffixes of X. It is essentially a Compressed Trie built on the set of suffixes $S = \{X_{1..n}, X_{2..n}, ..., X_{n..n}\}$.

Definition 6.3. A Suffix Tree T for a string X of length n is a directed tree with exactly n leaves numbered 1 to n. Each internal node has at least two children. Each edge is labeled with a non-empty substring of X. No two edges leaving a node have labels starting with the same character. The concatenation of labels from the root to leaf i spells out the suffix $X[i..n]$.

To save space, edge labels are stored as pairs of indices (i, j) representing $X[i..j]$.

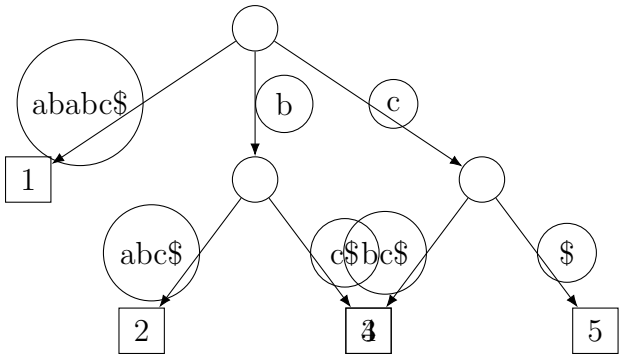

Figure 6.3: Figure 6.6: Suffix Tree for $X = ababc$ (conceptually). Indices (i, j) are used in implementation.

6.2.1 Construction and Applications

A Suffix Tree can be built in $O(n)$ time using algorithms by Weiner (1973), McCreight (1976), or Ukkonen (1995).

Applications:

- **Pattern Matching:** To check if pattern P exists in X, traverse the tree from the root matching characters of P. Time: $O(|P|)$.
- **Longest Repeated Substring:** Find the deepest internal node (measured by string depth). This node represents the longest substring appearing at least twice. Time: $O(n)$.
- **Longest Common Substring:** Build a generalized suffix tree for two strings X and Y. Find the deepest node that has leaf descendants from both strings. Time: $O(|X| + |Y|)$.

6.3 The Interpolation Search Tree (IST)

The IST is a tree structure supporting Interpolation Search in a set S ($|S| = n$) with average access time $O(\log \log n)$. It also supports dynamic updates efficiently.

Properties:

1. Space: $O(n)$.
2. Amortized Insert/Delete: $O(\log n)$.
3. Average Search Time: $O(\log \log n)$ for smooth distributions.
4. Worst-case Search Time: $O((\log n)^2)$.

Structure: An IST node is a multi-way node. The root has degree $\Theta(\sqrt{n})$, dividing the file into $\sqrt{n}$ subfiles. Each node contains:

- **REP Array:** Representatives of the subtrees.
- **ID Array:** An approximation of the inverse distribution function, used to interpolate the position of an element.

Operations: Insertions and Deletions are handled by rebuilding subtrees when they become unbalanced or "counters" overflow. The reconstruction is based on the idea that an ideal IST (perfectly balanced) can be built in $O(n)$ time.

Search Algorithm: To search for y: 1. Use the ID array to estimate the position j in the REP array. 2. Scan linearly from j to find the correct subtree. 3. Recursively search in the child node.

Theorem 6.12. For a smooth probability density μ, the average search time in a random IST of size n is $O(\log \log n)$.

Chapter 7

Structures Based on Representation

In the previous chapters, we examined data structures that rely on comparisons between elements (Comparison-based). In this chapter, we will examine structures that use the digital representation of the elements (e.g., their binary value) to store and retrieve them efficiently. These structures are often called **representation-based** structures.

7.1 Hashing

Let U be a large universe of keys (e.g., all possible names), and $S \subseteq U$ be the set of n keys we actually want to store. Since $|U| \gg n$, using a direct addressing table of size $|U|$ is impractical. Instead, we use a **Hash Table** T of size m (where $m \approx n$) and a function $h : U \to \{0, 1, ..., m-1\}$, called a **hash function**. An element x is stored in slot $T[h(x)]$.

Collision: A collision occurs when two distinct keys $x, y \in S$ map to the same slot, i.e., $h(x) = h(y)$.

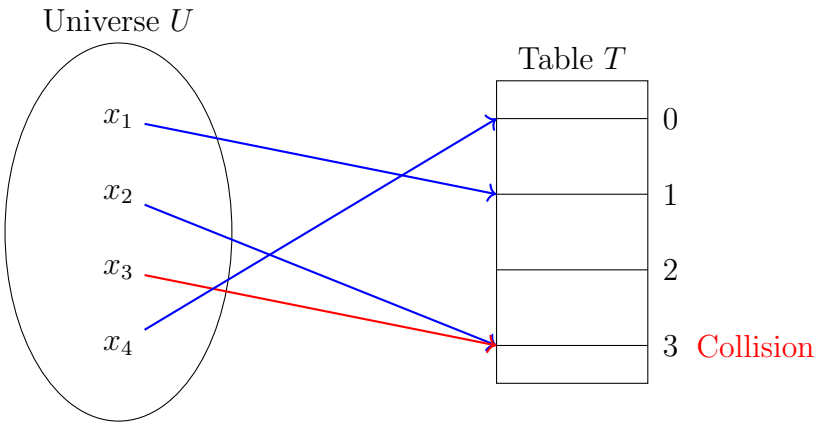

Figure 7.1: Figure 7.1: Hashing keys to a table. x_2 and x_3 collide.

7.1.1 Chained Hashing

In **Chained Hashing**, each slot $T[j]$ contains a pointer to a linked list of all elements that hash to j.

Operations:

- `Insert(x)`: Compute $h(x)$ and insert x at the head of the list $T[h(x)]$. Time: $O(1)$.

- `Search(x)`: Compute $h(x)$ and search the list $T[h(x)]$. Time: Proportional to the list length.
- `Delete(x)`: Compute $h(x)$, search for x in the list, and remove it.

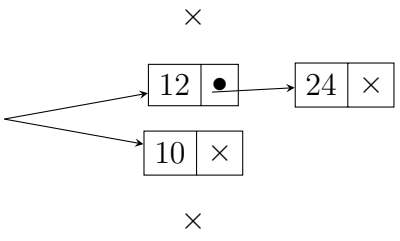

Figure 7.2: Figure 7.4: Chained Hashing. Slot 1 contains keys 12 and 24.

7.1.2 Average Case Analysis

We define the **load factor** $\alpha = n/m$. Under the assumption of **Simple Uniform Hashing** (any element is equally likely to hash into any of the m slots), the average length of a list is α. **Theorem:** An unsuccessful search takes average time $\Theta(1+\alpha)$. A successful search takes average time $\Theta(1+\alpha)$. If $m \propto n$, then $\alpha = O(1)$, and operations take $O(1)$ on average.

7.1.3 Open Addressing

In **Open Addressing**, all elements are stored directly in the table. If a collision occurs, we probe for the next available slot using a probe sequence. The hash function becomes $h(x, i)$, where i is the probe number $(0, 1, ...)$.

Common probing techniques:

1. **Linear Probing:** $h(x,i) = (h'(x) + i) \pmod m$.
2. **Quadratic Probing:** $h(x,i) = (h'(x) + c_1 i + c_2 i^2) \pmod m$.
3. **Double Hashing:** $h(x,i) = (h_1(x) + i h_2(x)) \pmod m$.

Double Hashing is generally the best method as it produces permutations that are closer to random.

7.2 Advanced Hashing Techniques

7.2.1 Perfect Hashing

If the set S is static (known in advance), we can construct a hashing scheme that guarantees **zero collisions** in the worst case. This usually involves a two-level hashing scheme where the second level uses a hash function tailored to the specific keys colliding at that bucket.

7.2.2 Universal Hashing

To avoid the worst-case scenario where an adversary chooses keys that all hash to the same slot, we select the hash function **randomly** from a carefully designed class of functions. This guarantees that for any distinct keys x, y, the probability of a collision is at most $1/m$.

7.2.3 Extendible Hashing

When the database is too large to fit in main memory and is stored on disk, we use Extendible Hashing. It uses a directory (index) in main memory and buckets (pages) on the disk.

- The directory uses the first d bits (global depth) of the hash to point to buckets.
- When a bucket overflows, it splits, and the directory size may double if necessary.

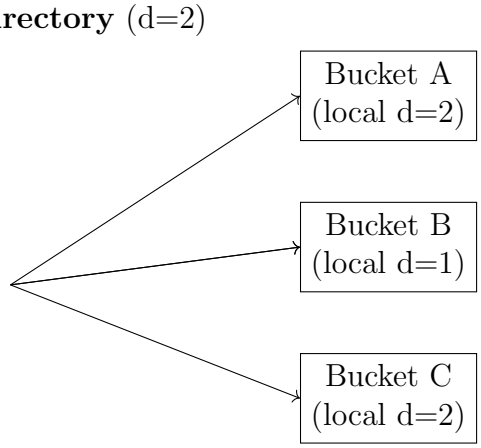

Figure 7.3: Figure 7.9: Extendible Hashing. Directory entries point to disk blocks.

7.3 Structures for the RAM Model

The RAM model allows arithmetic and bitwise operations on keys in constant time, enabling structures that beat the comparison-based lower bounds.

7.3.1 The van Emde Boas Tree

A **van Emde Boas (vEB) tree** stores a subset of the universe $U = \{0, ..., 2^k - 1\}$. It supports operations in $O(\log \log N)$ time, where N is the universe size. The structure works recursively:

- The universe of size u is split into $\sqrt{u}$ clusters of size $\sqrt{u}$.
- A key x is stored as a pair $\langle high(x), low(x) \rangle$, where $high(x) = x/\sqrt{u}$ (cluster index) and $low(x) = x \pmod{\sqrt{u}}$ (offset within cluster).
- A summary structure keeps track of which clusters are non-empty.

7.3.2 The Fusion Tree

The **Fusion Tree** (Fredman & Willard) allows searching in $O(\log n / \log \log n)$ time on a RAM machine with word size w. It uses bitwise tricks to "fuse" multiple keys into a single machine word, allowing simultaneous comparisons.

Chapter 8

Persistent Data Structures

8.1 Introduction

Standard data structures are **ephemeral**, meaning that making changes destroys the old version of the structure, leaving only the new one. In many areas, such as computational geometry, text processing, and high-level programming languages, it is necessary to maintain multiple versions of a structure.

We call a data structure **persistent** if it supports access to multiple versions of itself.

- **Partially Persistent:** All versions can be accessed, but only the newest version can be modified.
- **Fully Persistent:** Every version can be both accessed and modified (creating a new version branch).

We assume a **Linked Data Structure**, which is a collection of nodes containing information fields and a constant number of pointers. Access is provided via entry pointers. Operations are of two types:

1. **Access:** Reaching a node via pointers.
2. **Update:** Changing a field or a pointer, or creating a new node.

8.2 Partial Persistence

Two general techniques are proposed for transforming linked data structures into partially persistent ones: the **Fat Node** method and the **Node Copying** method.

8.2.1 The Fat Node Method

In the Fat Node method, we assume that each node can store an arbitrary history of modifications. Instead of overwriting a field, we store the new value along with a timestamp (version number).

To access a field in version i, we search the node's history for the version $v \leq i$ closest to i. If there are m versions, this search takes $O(\log m)$ time (using binary search on the version history). The space overhead is $O(1)$ per modification.

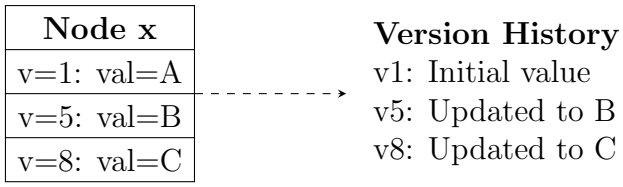

Figure 8.1: The Fat Node concept. The node stores a list of values paired with version timestamps.

8.2.2 The Node Copying Method

The Node Copying method addresses the $O(\log m)$ access time overhead of the Fat Node method. Here, each node has a fixed size but contains a small number of "extra" modification slots.

- When a field needs to be updated, if there is an empty extra slot, we store the new value and the version there.
- If the node is full (no empty slots), we **copy** the node to a new node containing only the latest values.
- We must then update the parent pointers to point to the new copy. This may trigger a cascade of copying up to the root (path copying).

Using amortized analysis, it can be proven that if nodes allow a sufficient constant number of extra slots (e.g., 2), the amortized cost of an update is $O(1)$, and the access time remains $O(1)$ because we don't need to search long histories.

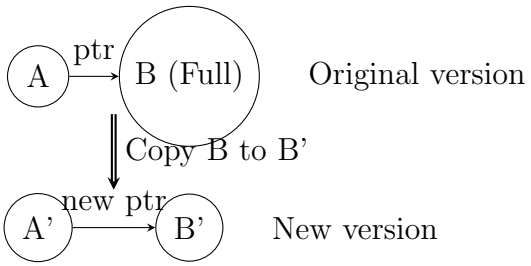

Figure 8.2: Node Copying. When B is full and modified, a copy B' is created, and A must be updated to point to B'.

Theorem 8.1. A linked data structure with constant in-degree can be made partially persistent with:

- **Fat Node:** $O(\log m)$ access time overhead, $O(1)$ update space/time.
- **Node Copying:** $O(1)$ access time, $O(1)$ amortized update space/time.

8.3 Full Persistence

In full persistence, versions form a **Version Tree** rather than a linear list, because we can update any past version, creating a branch in history. To navigate this tree efficiently, we can use a linearization technique (e.g., Euler tour) to map the tree structure to a linear order, allowing us to determine ancestry relationships.

The techniques used are extensions of the partial persistence methods:

1. **Fat Node:** Similar to partial persistence, but managing the version tree requires more complex comparisons. Access overhead becomes $O(\log m)$.

2. **Node Splitting:** This is the fully persistent equivalent of Node Copying. When a node is full, it is split into two nodes. The values are distributed between them.

Theorem 8.2. A linked data structure can be made fully persistent with $O(1)$ amortized space and update time, and $O(1)$ access time using the Node Splitting method.

Bibliographic Discussion

The techniques for persistence were formalized by Driscoll, Sarnak, Sleator, and Tarjan (1989). They cover general linked structures. Fiat and Kaplan (2003) provided solutions for confluent persistence (supporting merge operations).

Chapter 9

Priority Queues

A **Priority Queue** is an Abstract Data Type (ADT) that supports the following operations:

- `MakeQueue()`: Creates an empty queue.
- `Insert(Q, x)`: Inserts element x into Q.
- `FindMin(Q)`: Returns a pointer to the minimum element in Q.
- `DeleteMin(Q)`: Deletes the element with the minimum value.
- `Meld(Q1, Q2)`: Merges two queues Q_1 and Q_2.
- `DecreaseKey(Q, x, k)`: Decreases the key of node x to value k.
- `Delete(Q, x)`: Deletes node x.

We will examine two advanced implementations: **Binomial Heaps** and **Fibonacci Heaps**.

9.1 Binomial Queues (Heaps)

A Binomial Heap is a collection of **Binomial Trees**.

9.1.1 Binomial Trees

A Binomial Tree B_k is defined recursively:

- B_0 consists of a single node.
- B_k ($k \geq 1$) consists of two B_{k-1} trees, where the root of one becomes the leftmost child of the other.

Properties of B_k:

1. It has 2^k nodes.
2. Its height is k.
3. The root has degree k, and its children are $B_{k-1}, B_{k-2}, ..., B_0$.

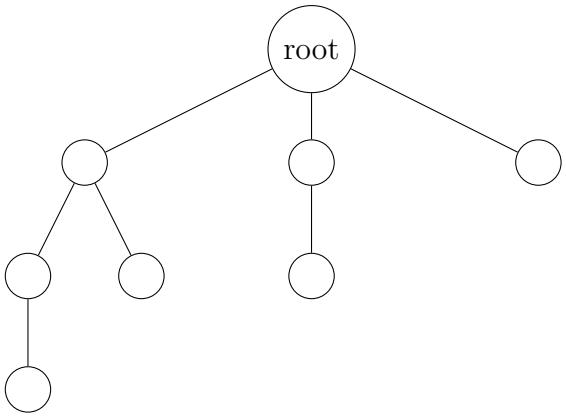

Figure 9.1: A Binomial Tree B_3. It is constructed by linking two B_2 trees.

9.1.2 Binomial Heaps

A Binomial Heap is a forest of Binomial Trees that satisfies:

1. **Heap Property:** For every node, its value is smaller than or equal to the values of its children.

2. There is at most one B_k for any degree k. This corresponds to the binary representation of the number of elements n.

Operations:

- **Meld:** Similar to binary addition. If both heaps contain a B_k, they are linked to form a B_{k+1}. Time: $O(\log n)$.

- **Insert:** Treated as a Meld of the existing heap with a new heap containing only B_0. Time: $O(\log n)$.

- **DeleteMin:** Find the min (a root), remove it, and meld its children (which form a new heap) with the remaining trees. Time: $O(\log n)$.

9.2 Fibonacci Heaps

Fibonacci Heaps improve the amortized complexity of operations. They are a collection of heap-ordered trees but do not enforce the strict structure of Binomial Heaps (lazy approach).

9.2.1 Representation

The roots of the trees are stored in a circular doubly linked list. We maintain a pointer `min` to the root with the minimum key. Nodes have a **mark** bit. A node is marked if it has lost a child since it became a child of its current parent.

9.2.2 Operations and Analysis

- **Insert/Meld:** Simply concatenate the root lists. Amortized Time: $O(1)$.

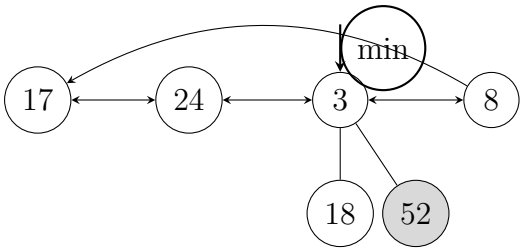

Figure 9.2: Fibonacci Heap structure. Roots are in a circular list. Marked nodes are shaded.

- **DeleteMin:** Remove min, promote its children to the root list. Then, **consolidate** trees so that no two roots have the same degree. Amortized Time: $O(\log n)$.
- **DecreaseKey:** Decrease the value. If the heap property is violated, cut the node and move it to the root list. If the parent was unmarked, mark it. If the parent was already marked, cut the parent as well (cascading cut). Amortized Time: $O(1)$.

9.3 Comparisons

Operation	**Binomial Heap (Worst Case)**	**Fibonacci Heap (Amortized)**
MakeQueue	$O(1)$	$O(1)$
Insert	$O(\log n)$	$O(1)$
FindMin	$O(\log n)$ ($O(1)$ if ptr stored)	$O(1)$
DeleteMin	$O(\log n)$	$O(\log n)$
Meld	$O(\log n)$	$O(1)$
DecreaseKey	$O(\log n)$	$O(1)$
Delete	$O(\log n)$	$O(\log n)$

Table 9.1: Complexity comparison.

Chapter 10

Union - Find

The Union-Find problem involves maintaining a partition of a universe $U = \{0, 1, ..., N-1\}$ into disjoint sets. Supported operations:

- **Find(x):** Returns the name of the set containing x.
- **Union(A, B):** Merges sets named A and B into a new set.

10.1 Efficient Find (Array Implementation)

We use an array `ID[0..N-1]` where `ID[i]` stores the set name of element i.

- **Find(x):** Returns `ID[x]`. Time: $O(1)$.
- **Union(A, B):** We must scan the array and change all entries of A to B (or vice-versa). Time: $O(N)$.

Optimization: Always rename the smaller set (**Weighted Union Rule**). A sequence of m operations takes $O(m + n \log n)$.

10.2 Efficient Union (Tree Implementation)

We represent each set as a tree. Each node points to its parent. The root represents the set name.

- **Union(A, B):** Make the root of one tree point to the root of the other. Time: $O(1)$.
- **Find(x):** Traverse up from x to the root. Time: Proportional to depth.

10.2.1 Optimizations

To improve `Find`, we use two heuristics:

1. **Weighted Union Rule:** Always link the root of the smaller tree (by size or rank) to the root of the larger tree. This ensures height is $O(\log n)$.
2. **Path Compression:** During `Find(x)`, make all nodes on the path from x to the root point directly to the root.

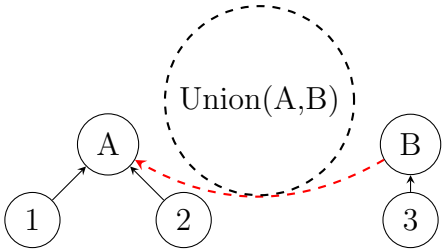

Figure 10.1: Tree representation. Union makes one root a child of the other.

Algorithm 12 Find(x) with Path Compression

```
1: if parent[x] ≠ x then
2:     parent[x] ← Find(parent[x]);
3: end if
4: return parent[x];
```

Theorem 10.3. A sequence of m operations on n elements using both heuristics takes $O(m \cdot \alpha(m, n))$ time, where α is the inverse Ackermann function. Since $\alpha(m, n) \leq 4$ for all practical purposes, the time is effectively linear.

Bibliography

[1] G. M. Adel'son-Vel'skiĭ and E. M. Landis. An algorithm for the organization of information. *Soviet Mathematics Doklady*, 3:1259–1263, 1962.

[2] A. Andersson and M. Thorup. Tight(er) worst case bounds for dynamic searching and priority queues. In *32nd Annual ACM Symposium on Theory of Computing (STOC2000)*, pages 335–342, 2000.

[3] L. Arge and J.S. Vitter. Optimal dynamic interval management in external memory. *SIAM Journal of Computing*, to appear, 2003.

[4] P. Bachmann. *Analytische Zahlentheorie*. Teubner, 1894.

[5] P. Beame and F. Fich. Optimal bounds for the predecessor problem and related problems. *Journal of Computer and System Sciences*, 65:38–72, 2002.

[6] M. Blum, R.W. Floyd, V. Pratt, R.L. Lewis, and R. E. Tarjan. Time bounds for selection. *Journal of Computer and System Sciences*, 7:448–461, 1973.

[7] N. Blum and K. Mehlhorn. On the average number of rebalancing operations in weight-balanced trees. *Theoretical Computer Science*, 11:303–320, 1980.

[8] G. Brodal, G. Lagogiannis, Ch. Makris, A. Tsakalidis, and K. Tsihlas. Optimal finger search trees in the pointer machine. In *34th Annual ACM Symposium on Theory of Computing (STOC)*, pages 583–591, 2002. Full version to appear in *Journal of Computer and System Sciences*.

[9] G. S. Brodal. Worst case efficient priority queues. In *ACM-SIAM Symposium on Discrete Algorithms (SODA96)*, pages 52–58, 1996.

[10] G.S. Brodal. Finger search trees with constant insertion time. In *9th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA98)*, pages 540–549, 1998.

[11] J.L. Carter and M.N. Wegman. Universal classes of hash functions. *Journal of Computer and System Sciences*, 18:143–154, 1979.

[12] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms*. The MIT Press, McGraw-Hill Book Company, 2nd edition, 2001.

[13] Martin Dietzfelbinger, Anna Karlin, Kurt Mehlhorn, Friedhelm Meyer Auf Der Heide, Hans Rohnert, and Robert E. Tarjan. Dynamic perfect hashing: Upper and lower bounds. *SIAM Journal of Computing*, 23:738–761, 1994.

[14] P. Dietz and R. Raman. A constant update time finger search tree. *Information Processing Letters*, 52:147–154, 1994.

[15] Paul F. Dietz and Daniel Dominic Sleator. Two algorithms for maintaining order in a list. In *19th Annual ACM Symposium on Theory of Computing (STOC87)*, pages 365–372, 1987.

[16] J. R. Driscoll, N. Sarnak, D. Sleator, and R.E. Tarjan. Making data structures persistent. *Journal of Computer and System Sciences*, 38(1):86–124, 1989.

[17] R. Fagin, J. Nievergelt, N. Pippenger, and H.R. Strong. Extendible hashing: a fast access method for dynamic files. *ACM Transactions on Database Systems*, 4:315–344, 1979.

[18] Amos Fiat and Haim Kaplan. Making data structures confluently persistent. *Journal of Algorithms*, 48(1):16–58, 2003.

[19] Ph. Flajolet and J.M. Steyaert. A branching process arising in dynamic hashing, trie searching and polynomial factorization. In *9th Intl. Colloquium on Automata, Languages and Programming (ICALP82)*, LNCS, pages 239–251. Springer Verlag, 1982.

[20] Lestor R. Ford, Jr., and Selmer M. Johnson. A tournament problem. *The American Mathematical Monthly*, 66:387–389, 1959.

[21] G. Franceschini and R. Grossi. Optimal space-time dictionaries over an unbounded universe with at implicit trees. Technical Report TR-03-03, Università di Pisa, 2003.

[22] G. Franceschini, R. Grossi, J. Ian Munro, and L. Pagli. Implicit b-trees: New results for the dictionary problem. In *ACM-IEEE Foundations of Computer Science (FOCS2002)*, pages 145–154, 2002.

[23] E. Fredkin. Trie memory. *Communications of the ACM*, 3:490–499, 1962.

[24] Michael L. Fredman and Robert E. Tarjan. Fibonacci heaps and their uses in improved network optimization algorithms. *Journal of the ACM*, 34(3):596–615, 1987.

[25] M.L. Fredman, J. Komlós, and E. Szemerédi. Storing a sparse table with O(1) worst case access time. *Journal of the ACM*, 31:538–544, 1984.

[26] M.L. Fredman and D.E. Willard. Surpassing the information theoretic bound with fusion trees. *Journal of Computer and System Sciences*, 47:424–436, 1993. (First appeared in STOC90).

[27] Harold N. Gabow and Robert E. Tarjan. A linear-time algorithm for a special case of disjoint set union. *Journal of Computer and System Sciences*, 30(2):209–221, 1985.

[28] Zvi Galil and G.F. Italiano. Data structures and algorithms for disjoint set union problems. *ACM Computing Surveys*, 23(3):319–344, 1991.

[29] A.M. Garsia and M.L. Wachs. A new algorithm for minimum cost binary trees. *SIAM Journal of Computing*, 4:622–642, 1977.

[30] Leonidas J. Guibas and Robert Sedgewick. A dichromatic framework for balanced trees. In *IEEE Conference on Foundations of Computer Science (FOCS1978)*, pages 8–21. IEEE Computer Society, 1978.

[31] L.J. Guibas, E.M. McCreight, M.P. Plass, and J.R. Roberts. A new representation for linear lists. In *9th Annual ACM Symposium on Theory of Computing (STOC77)*, pages 49–60, 1977.

[32] Y. Han. Improved fast integer sorting in linear space. In *34th Annual ACM Symposium on Theory of Computing (STOC2002)*, pages 602–608, 2002.

[33] C.A.R. Hoare. Algorithm 63 partition and algorithm 65 find. *Communications of the ACM*, 4:321–322, 1961.

[34] C.A.R. Hoare. Quicksort. *Computer Journal*, 5:10–15, 1962.

[35] S. Huddleston and K. Mehlhorn. A new data structure for representing sorted lists. *Acta Informatica*, 17:157–184, 1982.

[36] T.C. Hu and A. Tucker. Optimum computer search trees. *SIAM Journal of Applied Mathematics*, 21:514–532, 1971.

[37] Haim Kaplan and Robert E. Tarjan. Purely functional, real-time deques with catenation. *Journal of the ACM*, 46(5):577–603, 1999.

[38] Alexis Kaporis, Christos Makris, Spyros Sioutas, Athanasios Tsakalidis, Kostas Tsihlas, and Christos Zaroliagis. Improved bounds for finger search on a RAM. In *11th Annual European Symposium on Algorithms (ESA2003)*, 2003.

[39] G. D. Knott. Expandable open addressing hash table storage and retrieval. In *ACM SIGFIDET Workshop on Data Description, Access and Control*, pages 182–206, 1971.

[40] Donald E. Knuth. *Fundamental Algorithms*, volume 1 of *The Art of Computer Programming*. Addison-Wesley, 1997.

[41] Donald E. Knuth. *Sorting and Searching*, volume 3 of *The Art of Computer Programming*. Addison-Wesley, 1997.

[42] P.A. Larson. Dynamic hashing. *BIT*, 18:184–201, 1978.

[43] W. Litwin. Virtual hashing: A dynamically changing hashing. In *ACM Conf. on Very Large Databases (VLDB78)*, pages 517–523, 1978.

[44] E. McCreight. A space-economical suffix tree construction algorithm. *Journal of the ACM*, 23:262–272, 1976.

[45] K. Mehlhorn and A. Tsakalidis. Data structures. In *Algorithms and Complexity*, edited by J. van Leeuwen, volume A, chapter 6, pages 301–341. Elsevier, 1990.

[46] K. Mehlhorn and A.K. Tsakalidis. An amortized analysis of insertions into AVL-trees. *SIAM Journal of Computing*, 15:22–33, 1986.

[47] K. Mehlhorn and A.K. Tsakalidis. Dynamic interpolation search. *Journal of the ACM*, 40:621–634, 1993.

[48] Kurt Mehlhorn. *Sorting and Searching*, volume 1 of *Data Structures and Algorithms*. Springer Verlag, 1984.

[49] J. Ian Munro. An implicit data structure supporting insertion, deletion, and search in $O(\log^2 n)$ time. *Journal of Computer and System Sciences*, 33:66–74, 1986.

[50] J. Ian Munro and H. Suwanda. Implicit data structures for fast search and update. *Journal of Computer and System Sciences*, 21:236–250, 1980.

[51] J. Nievergelt and E.M. Reingold. Binary search trees of bounded balance. *SIAM Journal of Computing*, 2:33–43, 1973.

[52] H.J. Olivié. A new class of balanced search trees: Half-balanced search trees. *R.A.I.R.O. Informatique Théorique*, 16(1):51–71, 1982.

[53] Th. Ottman and H.W. Six. Eine neue klasse von ausgeglichenen binärbäumen. *Angewandte Informatik*, 18:395–400, 1976.

[54] J. Papatheodorou. *Algorithms*. University of Patras Press, 2000.

[55] Y. Pearl, A. Itai, and H. Avni. Interpolation search – a $\log\log n$ search. *Communications of the ACM*, 21:550–554, 1978.

[56] Y. Pearl and E.M. Reingold. Understanding the complexity of interpolation search. *Information Processing Letters*, 6:219–222, 1977.

[57] J.A. La Poutré. Lower bounds for the union-find and split-find on pointer machines. *Journal of Computer and System Sciences*, 52:87–99, 1996.

[58] R. E. Schönhage. Storage modification machines. *SIAM Journal on Computing*, 9:490–508, 1980.

[59] Daniel D. Sleator and Robert E. Tarjan. The amortized efficiency of list update and paging rules. *Communications of the ACM*, 28(2):202–208, 1985.

[60] Daniel D. Sleator and Robert E. Tarjan. Self-adjusting binary search trees. *Journal of the ACM*, 32(3):652–686, 1985.

[61] M. Tamminen. Extendible hashing with overflow. *Information Processing Letters*, 15:227–233, 1982.

[62] R. E. Tarjan. A class of algorithms which require non-linear time to maintain disjoint sets. *Journal of Computer and System Sciences*, 18:110–127, 1979.

[63] R. E. Tarjan. Amortized computational complexity. *SIAM Journal on Algebraic and Discrete Methods*, 6:306–318, 1985.

[64] Robert E. Tarjan. Updating a balanced search tree in O(1) rotations. *Information Processing Letters*, 16:253–257, 1983.

[65] Robert E. Tarjan. Sequential access in splay trees takes linear time. *Combinatorica*, 5(4):367–378, 1985.

[66] Robert E. Tarjan and J. van Leeuwen. Worst-case analysis of set union algorithms. *Journal of the ACM*, 31(2):245–281, 1984.

[67] M. Thorup. Equivalence between priority queues and sorting. In *43rd Symposium on Foundations of Computer Science (FOCS2002)*, pages 135–144, 2002.

[68] Athanasios Tsakalidis. *Data Structures*. University Lecture Notes, 2002.

[69] A. K. Tsakalidis. Maintaining order in a generalized linked list. *Acta Informatica*, 21:101–112, 1984.

[70] A. K. Tsakalidis. AVL trees for localized search. *Information and Control*, 67:173–194, 1985.

[71] A. K. Tsakalidis. Rebalancing operations for deletions in AVL-trees. *R.A.I.R.O. Informatique Théorique*, 19:323–329, 1985.

[72] E. Ukkonen. On-line construction of suffix trees. *Algorithmica*, 14:249–260, 1995.

[73] P. van Emde Boas. Preserving order in a forest in less than logarithmic time and linear space. *Information Processing Letters*, 6:80–82, 1977.

[74] P. van Emde Boas, R. Kaas, and E. Zijlstra. Design and implementation of an efficient priority queue. *Mathematical Systems Theory*, 10:99–127, 1977.

[75] Jean Vuillemin. A data structure for manipulating priority queues. *Communications of the ACM*, 21(4):309–315, 1978.

[76] P. Weiner. Linear pattern matching algorithms. In *14th IEEE Symposium on Switching and Automata Theory*, pages 111, 1973.

[77] D.E. Willard. Log-logarithmic worst case range queries are possible in space $\Theta(N)$. *Information Processing Letters*, 17:81–84, 1983.

[78] D.E. Willard. New trie data structures which support very fast search operations. *Journal of Computer and System Sciences*, 28:379–394, 1984.

[79] D.E. Willard. Searching unindexed and nonuniformly generated files in $\log\log N$ time. *SIAM Journal of Computing*, 14:1014–1029, 1985.

[80] J.W.J. Williams. Algorithm 232: Heapsort. *Communications of the ACM*, 7:347–348, 1964.