Data Organization

Martino Papa

19 march 2025

Contents

Introduction

In 2020 the *Digital Universe Study of International Data Corporation* estemed that humanity have stored 40ZB of data, corrisponding to 5247 GB per person. Only 1% of those data has been analysed due to the high presence of desctructureted data. **Big data analytics** consists in using sophisticated techniques on a big amount of data with the goal of describing events, situations, identyfing patterns and finding correaltions. There are 4 different types of data analytics:

- descreptive analysis, understanding what happend;
- predictive analysis, predicting what's going to happend;
- prescreptive analysis, understanding how to react to a certain event;
- diagnostical analysis, understanding why something happend.

In the Big data analytics there are 3 main problems known as the "three V".

- volume, the enormus amount of data bring problems to memorization and analysis;
- velocity, the amount of new data makes them difficoult to process online;
- varaiety, different types of data are difficoult to handle.

Chapter 1

Memory hierarchy

In a computer the data can be stored in different types of memories with different goals. In order to be processed from the CPU data needs to be in the CPU registers, whose are fast but also very small thus we need to store all the data somplace else. There are different types of memories, the bigger they get the slower they will be. All the data are stored in the biggest one and then they are moved to the smaller ones when they need to be processed. The memories inside a computer listed from the slowest and biggest to the fastest and smallest are the following.

- External Memory (EM)
- Random Access Memory (RAM)
- Cache
- CPU registers

Usually in algorithms complexity analysis all memory access are treated as they would take the same time. In fact if an algorithm has to work on different machines we cannot refer to a specific structure. To imporve the performance Operating Systems choose where to store the data following two principles.

Proposition 1.1 (Locality of Time)**.** *If a program accesses certain memory locations, it is likely to access the same locations again in the near future.*

Proposition 1.2 (Locality of Space)**.** *If a program accesses a particular memory location, it is likely to access nearby memory locations soon after.*

When we move data two the faster memories we count on the fact that Locality of Time tells that they are probably gonna be needed again. To profit by Locality of Space the data are moved in blocks.
When an access to a location l is required the computer proceeds as follows:

▷ the CPU determines if l is in the cache
if l is in the cache **then**
    ▷ the CPU access l and goes two the next instruction
else
    ▷ the block b of EM containing l is moved in the cache
end if

When a block b is moved if the cache is already full one block has to be removed in regard to load b. To choose which block two remove there are differents possible strategies.

- **Random choice**. Easy to implement, the cost is constant but locality of time and space aren't used.
- **First In First Out (FIFO)**. Easy to implement with a queue, the cost is constant, tries to profit by Locality of Time.
- **Least Frequently Used (LFU)**. Implemented with a priority queue based on the number of accesses to every block. Tries to profit by both Time and Space Locality but it penalizes recently added blocks.

- **Least Recently Used (LRU)**. Implemented with a priority queue with locators (references that allow efficient accesses). High implementation cost. Very efficient, priors mostly by Locality of Time.
- **Marker**. A boolean value **marked** initialized to *False* is assigned to each block in the cache, when the program access a block it sets **marked**=*True*. When a block needs to be removed a random one having **marked**=*False* is choosen. Once every block in the cache has value *True* they all change to *False*.

Competitive analysis of the strategies To analyse how good a strategy is we compare it to a theoretical optimal one.

Definition 1.1 (On-line algorithm)**.** *They perform on an unknown set of requests with different costs.*

Definition 1.2 (Off-line algorithm)**.** *They perform on a pre-known set of requests.*

Competitive analysis consists in confronting on-line algorithms with an optimal off-line one.

Definition 1.3 (c-competitivie algorithm)**.** *Let OPT be the optimal off-line algorithm, A an on-line algorithm, $P = (P_1, \ldots, P_n)$ a sequence of instructions. We say that A is c-competitivie, $c \geq 0$ if*

$$cost(A, P) \leq c \cdot cost(\mathsf{OPT}, P) + b,\ b \geq 0 \tag{1.1}$$

c is defined as competitive factor of A.

Proposition 1.3. *FIFO and LRU strategies are m-competitive with m number of blocks in the cache.*

An on-line randomised algorithm can have different executions on the same sequence of instructions, in this case we say that is c-competitive if

$$\mathbb{E}[\text{cost}(A, P)] \leq c \cdot \text{cost}(\mathsf{OPT}, P) + b,\ b \geq 0 \tag{1.2}$$

Proposition 1.4. *Marker strategy has a competitive factor $O(\log m)$, with m beeing the number of blocks in the cache.*

1.1 Computational models

Definition 1.4 (Architectual model)**.** *The architectual model represents the essential components involved in the execution of algorithms.*

Definition 1.5 (Programming model)**.** *The programming model specifies the set of instructions used to describe algorithms.*

Definition 1.6 (Cost model)**.** *The cost model defines functions that mesure how good algorithms' performances are.*

The model that is usuallly used in analysis of algorithms and data structures is called **RAM model** and it's defined as follows.

- The architectual model consists in a RAM memory and a CPU.
- The programming model describes arithmetic, logic, reading, confronts and writing operations.
- The cost model defines the *time cost* as the number of operations that an algorithm requires, and the *space cost* as the number of used memory cells.

The model assumes that every reading and writing operation has a constant cost. This is not true with *big dimension problems*.

Definition 1.7 (Big dimension problem)**.** *A problem is called big dimension problem if the dimension of the input data is bigger than the dimension of the system's memory.*

To handle those problem we introduce an external memory. Those type of memory are much bigger but slower, the assumption that every reading and writing operation has a constant cost falls. Thus we replace the RAM model with a new model called **Disk model**.

- The architectual model consists in a CPU a RAM memory and an External Memory.
- The programming model describes, on top of the operation described in the RAM model, operations of input/output regarding the EM.
- The cost model defines two functions:
  - Work $W(N, M, B)$, number of elementar operations;
  - Transfer $T(N, M, B)$, number of I/O operations.

Where N is the dimension of the problem (number of datapoints), M the dimension of the internal memory (RAM), B dimension of transferd blocks (**pages**), Q number of interrogations to the memories and Z is the dimension of each answer. Referring to the number of needed blocks we also define

$$n = \frac{N}{B},\ m = \frac{M}{B},\ z = \frac{Z}{B}$$

With this notation we can analise how much fundamental operations cost in terms of Transfer.

- SCAN(N), scanning a file of N elements takes $\Theta(\frac{N}{B}) = \Theta(n)$ I/O operations.
- SORT(N), sorting a file of N elements takes $\Theta(n \log_m n)$ I/O operations.
- SEARCH(N), searching in file of N sorted elements takes $\Theta(\log_B n)$, $\Theta(1)$ with hash sets I/O operations.
- OUTPUT(Z), returning Z elemnts output of a query takes $\Theta(\max\{1, z\})$ output operations.

Chapter 2

Sorting on External Memories

The problem of sorting a set S consisting in N records memorised in External Memory can be firstly approached using the known algorithms used in central memory. Let's suppose that S is memorised in continuos postion of the EM, one of the most used algorithms in central memory is **Mergesort**.

MergeSort$(A, left, right)$

```
1: if left < right then
2:     mid ← ⌊(left + right)/2⌋
3:     MergeSort(A, left, mid)
4:     MergeSort(A, mid + 1, right)
5:     Merge(A, left, mid, right)
6: end if
```

Merge$(A, left, mid, right)$

```
1: n1 ← mid − left + 1, n2 ← right − mid
2: for i ← 1 to n1 do
3:     L[i] ← A[left + i − 1]
4: end for
5: for j ← 1 to n2 do
6:     R[j] ← A[mid + j]
7: end for
8: i ← 1, j ← 1, k ← left
9: while i ≤ n1 and j ≤ n2 do
10:    if L[i] ≤ R[j] then
11:        A[k] ← L[i]
12:        i ← i + 1
13:    else
14:        A[k] ← R[j]
15:        j ← j + 1
16:    end if
17:    k ← k + 1
18: end while
19: while i ≤ n1 do
20:    A[k] ← L[i]
21:    i ← i + 1, k ← k + 1
22: end while
23: while j ≤ n2 do
24:    A[k] ← R[j]
25:    j ← j + 1, k ← k + 1
26: end while
```

The algorithm divides recursively the set in two halfs untill it contains a single element, then the merge procedure joins two sorted half of dimension n_1, n_2 in $O(n_1+n_2)$ time. The algorithm works in $\Theta(n \log n)$

time and $\Theta(n)$ space. The problem is that it's not efficient in EM because it requires a lot of I/O operations. Mergesort is translated in the Disk Model with parameters M, B and N lenght of S by the algorithm 2-Mergesort defined as follows.

2-Mergesort(S, N, M, B)

1: **if** $N \leq M$ **then**
2: ▷ Sort S in RAM using traditionl Mergesort
3: **else**
4: Divide S in two parts S_1 and S_2
5: 2-MERGESORT(S_1, $N/2$, M, B)
6: 2-MERGESORT(S_2, $N/2$, M, B)
7: 2-Merge S_1 and S_2
8: **end if**

2-Merge(S_1, S_2, S)

1: **Load into RAM** the first block of S_1 and S_2: let them be S_{0_1} and S_{0_2}
2: $t \leftarrow |S_1|$
3: $B \leftarrow$ block size
4: $i \leftarrow 1$; $j \leftarrow 1$; $l \leftarrow 0$
5: Let β be an empty block in RAM
6: **while** $l < 2t$ **do**
7: $\beta \leftarrow B$ smallest keys between those of S_1 and S_2 present in RAM
8: Write β to disk as S_l
9: $l \leftarrow l + 1$
10: Empty β in RAM
11: **if** (there are fewer than B keys from S_1 in RAM) **and** $(i < t)$ **then**
12: **Load** S_{i_1} into RAM
13: $i \leftarrow i + 1$
14: **end if**
15: **if** (there are fewer than B keys from S_2 in RAM) **and** $(j < t)$ **then**
16: **Load** S_{j_2} into RAM
17: $j \leftarrow j + 1$
18: **end if**
19: **end while**

2-MergeSort copies the strategy adopted by Mergesort in RAM, but it uses the 2-Merge algorithm to merge two sorted blocks in EM. It's easy to notice that by doing so **the RAM isn't efficiently used**. The algorithm costs $T(N, M, B) = \Theta(\frac{N}{B} \log_2 \frac{N}{M})$ I/O operations and $W(N, M, B) = \Theta(N \log_2 N)$ elemntar operations. The number of I/O operations is improved by MB-MergeSort algorithm.

2.1 M/B MergeSort

Observing the Transfer complexity of 2-MergeSort is clear that to reduce the number of I/O operations we need to reduce the height of the recursion tree. To do so we can buy a bigger RAM or use the one we own in a better way. To do that instead of dividing the problem in 2 subproblems we divide it in $\frac{M}{B}$ subproblems.
The problem with MB-MergeSort is that now the Merge operation needs more than a compare operation to join two vectors since they're not sorted. To solve this problem use a **priority queue**. In this case the height of the recursion tree is the smallest h such that

$$\frac{N}{(M/B)^h} \leq M \text{ or equivalently } h \geq \log_{M/B} \frac{N}{M} \tag{2.1}$$

Theorem 2.1 (Transfer complexity of MB-MergeSort)**.** *The MB-MergeSort algorithm transfer complex-*

MB-MERGESORT(S)

$N \leftarrow |S|$
if $N \leq M$ **then**
    Solve the problem in RAM using traditional MergeSort
else
    for $i \leftarrow 1$ to M/B **do**
        Create S_i on disk, where:
        $S_i = S_{(i-1)\cdot\frac{N}{M/B}\ldots i\cdot\frac{N}{M/B}-1}$
        MB-MERGESORT(S_i)
    end for
    MB-MERGE($S_1, S_2, \ldots, S_{M/B}, S$)
end if

MB-MERGE($S_1, S_2, \ldots, S_{M/B}, S$)

Load into RAM $S_1^0, S_2^0, \ldots, S_{M/B}^0$
$t \leftarrow |S_1|/B$, $l \leftarrow 0$
Let β be an empty block in RAM
Create an empty priority queue Q in RAM
for $i \leftarrow 1$ to M/B **do**
    Insert the first key of S_i^0 into Q
end for
while not Q.isEmpty() **do**
    Extract the minimum key k from Q
    Insert k into the first available position in β
    if β contains B keys **then**
        Write β as S_l on disk, $l \leftarrow l + 1$
        Clear β in RAM
    end if
    Let $k \in S_i^j$ (i.e., k belongs to S_i^j)
    if all keys of S_i^j have been inserted into Q **then**
        if $j < t - 1$ **then**
            Load S_i^{j+1} into RAM
            Insert the first key of S_i^{j+1} into Q
        end if
    else
        Insert the next key of S_i^j into Q
    end if
end while

ity is expressed by the following recursion relation.

$$T(N,M,B) = \begin{cases} \Theta(\frac{N}{B}) & \text{if } N \leq M \\ \Theta(\frac{M}{B})T\left(\frac{N}{M/B}, M, B\right) + \Theta(\frac{N}{B}) & \text{if } N > M \end{cases} \tag{2.2}$$

This implies that the transfer complexity is $\Theta(\frac{N}{B}\log_{M/B}\frac{N}{M}) = \Theta(n\log_m n)$.

Theorem 2.2 (Work complexity of MB-MergeSort)**.** *The MB-MergeSort algorithm work complexity is expressed by the following recursion relation.*

$$W(N,M,B) = \begin{cases} \Theta(N\log_2 N) & \text{if } N \leq M \\ \Theta(\frac{M}{B})W\left(\frac{N}{M/B}, M, B\right) + \Theta(N\log_2\frac{M}{B}) & \text{if } N > M \end{cases} \tag{2.3}$$

This implies that the work complexity is $\Theta(N\log_2 N)$.

2.1.1 Compare with 2-MergeSort

The job done by the two algorithms at each recursion level is

$$\text{2-MergeSort:}\quad W(N,M,B) = \Theta(N),\ T(N,M,B) = \Theta(\frac{N}{B})$$

$$\text{MB-MergeSort:}\quad W(N,M,B) = \Theta(N\log_2\frac{M}{B}),\ T(N,M,B) = \Theta(\frac{N}{B})$$

The higher work done by MB-MergeSort is due by the fact that in the merge operation the two input vectors are not sorted. MB-MergeSort is prefered because since he has less recursion level the total number of operation has the same complexity of 2-MergeSort but with less I/O operations, which are the expensive ones.

Chapter 3

Searching on External Memories

Before jumping into how to perform the reserch in EM let's analyse how to do it in RAM. If the data are stored in a sequential, not-sorted structure (like vectors) the search can be done in $\Theta(n)$ by simply scanning all the vector in search of the desired element. This process is unefficient, in fact, if the data are sorted we can use the **Binary Search** algorithm that works in $\Theta(\log n)$ time.

BinarySearch($key, left, right, v$)

```
1: if right < left then
2:     return −1
3: end if
4: m ← ⌊(left + right)/2⌋
5: if key < v[m] then
6:     return BinarySearch(key, left, m − 1, v)
7: else if key > v[m] then
8:     return BinarySearch(key, m + 1, right, v)
9: else
10:    return m
11: end if
```

This algorithm works by splitting the vector in two halfs and checking if the key is in the left or right half. This gives the idea of using a binary trees to store the data.

Definition 3.1 (Binary Search Tree (BST))**.** *A Binary Search Tree is a binary tree in which each node has an associated key and satisfies the following property. For every node in the BST, the key stored in the node is* ***greater*** *than all the keys stored in the nodes of its* ***left subtree*** *and* ***smaller*** *than all the keys stored in the nodes of its* ***right subtree****.*

The advantage of using sorted structures is that we can perform the search in $\Theta(\log n)$ time. Of couse tough, inserting and removing items from those structures is more expensive than in unsorted structures. Furthermore in binary trees if we want to grant the effficency of the search we need to balance the tree. The most used balanced binary tree is the **AVL Tree**.

Definition 3.2 (AVL Tree)**.** *An AVL Tree is a binary search tree in which for every node the difference between the height of the left and right subtree is at most 1. In AVL trees the height of the tree is* $\Theta(\log n)$ *and adding or removing an element takes* $\Theta(\log n)$ *time.*

Another structure that grants the balancement of the tree is the **2-3 Tree**.

Definition 3.3 (2-3 Node)**.** *A 2-3 Node is a sorted set containing two keys* $k_1 < k_2$ *and three pointers* p_0, p_1, p_2*.*

- p_0 *points to the subtree containing keys smaller than* k_1*;*
- p_1 *points to the subtree containing keys between* k_1 *and* k_2*;*
- p_2 *points to the subtree containing keys greater than* k_2*.*

Definition 3.4 (2-3 Tree)**.** *A 2-3 Tree is a set of 2-3 nodes such that a root node A exists with the property that, for each other node B, there's only a path connecting A and B. 2-3 Trees also satisfie the following properties.*

- *All the leaves are at the same level;*
- *Every internal node (not leaf) has 2 or 3 children;*

2-3 Tree allow to reduce the height of the tree.

Theorem 3.1. *The height h of a 2-3 Tree with n keys satisfies*

$$0.631 \log_2(n+1) \leq h \leq \log_2(n+1) \tag{3.1}$$

Proof. The maximum height of a 2-3 Tree is found when every node contains only one key (equivalent to a binary tree). In this case the i-th level contains 2^i nodes so

$$\sum_{i=0}^{h-1} 2^i \leq N \Rightarrow 2^h - 1 \leq N \Rightarrow h \leq \log_2(N+1) \tag{3.2}$$

The minimum height of a 2-3 Tree is found when every node contains two keys. In this case the i-th level contains 3^i nodes so

$$\sum_{i=0}^{h-1} 2 \cdot 3^i \geq N \Rightarrow 2\frac{3^h - 1}{3-1} = 3^h - 1 \geq N \tag{3.3}$$

Thus we have using the logarithm properties:

$$h \geq \log_3(N+1) \Rightarrow h \geq \frac{\log_2(N+1)}{\log_2(3)} = 0.631 \log_2(N+1) \tag{3.4}$$

□

3.1 Problem description

We will suppose that the memory is divided in blocks of size B called **pages**, our goal is orgainize elements into pages to minimize the number of pages that we need to access. In fact I/O operations are much more expensive than elementar ones.

Definition 3.5 (Page)**.** *A page is defined as a contiguous block of B elements.*

We will also suppose the use of **Big Data Indexing** strategies wich allow to mantain dinamicly a large collection of data and offer methods to search, insert and delete elements efficiently. To do so we introduce the concept of key.

Definition 3.6 (Key)**.** *Given a collection $\mathcal{C}$ with attributes $A_1, \ldots, A_t$ a **key** for $\mathcal{C}$ is a subset of q attributes $A_{i_1}, \ldots, A_{i_q}$ on which a sorting order is defined. A key can be **primary** if it's unique for each element of $\mathcal{C}$ or **secondary** if it's not.*

Using the concept of key we can now give a formal definition of index.

Definition 3.7 (Index)**.** *An index is a searching external structure defined on collection $\mathcal{C}$ using a key K that implements efficiently the following procedures.*

- ***Searching** of the records having a certain key value*
- ***Insertion** of a new record in $\mathcal{C}$*
- ***Deletion** of a record*
- ***Range query** (optional), research of all the records having a key value in a certain range*
- ***Next** (optional), research of the records having the smallest key value greater than a certain value*

*An index is **primary** if the organization of the records is based on the organization of the key K, **secondary** if the records of $\mathcal{C}$ are mantained without any spacified order.*

3.2 Structures for index representation

Basic structures used to represent indexes are tree structures, hash tables, multidimensional indexes and bitmap. Firstly we will focus on the storage of **Binary Trees** on External Memory. There are different methods to do it, let's look into some of them.

3.2.1 Sequential method

It's the most basic method, it consists in adding every node to the same page untill it's full, then we move to the next page. This method is very simple but it's not efficient because it doesn't take advantage of the locality of space. On the bright side it's very easy to implement and it minimizes memory usage.

Exercise example 3.1. *Build a binary tree with the following keys: 51, 32, 28, 86, 40, 63, 58, 12, 90, 25, 27, 48, 93, 59, 33 and store it in pages of $B = 4$ nodes using the Sequential Method.*

Theorem 3.2 (I/O complexity)**.** *Given a tree with N nodes and pages containing B nodes the average number off accesses to pages done by the Sequential method is*

$$d_n = \frac{2N+B+1}{N}\log\frac{N}{B}+\frac{2B+1}{N}+\frac{N+B}{N}\gamma-1-\frac{(N+1)\cdot(B+1)}{NB}-\frac{0.068(N+1)B}{N(B+1)}+O\left(\frac{B^3}{N^3}\right) \quad (3.5)$$

Where the average is considered on all the possible $N!$ permutations of nodes, $\gamma = 0.57721$ eulero's constant.

3.2.2 Grouped method

The Grouped Method works as follows. When storing a node a if we can we store the a in the same page of his parent. If that page is full we store a in a new page. This method is more efficient than the previous one because it takes advantage of the locality of space. The problem with this method is that it doesn't guarantee that the tree will be balanced, thus we can have a lot of empty pages.

Exercise example 3.2. *Redo the exercise 3.1 using the Grouped Method.*

Theorem 3.3 (I/O complexity)**.** *Given a tree with N nodes and pages containing B nodes the average number of accesses to pages done by the Grouped method is*

$$d_n = \frac{H_N}{H_{B+1}-1} \quad (3.6)$$

Where the average is considered on all the possible $N!$ permutations of nodes and H_n is the n-th harmonic number defined as follows:

$$H_n = \sum_{i=1}^{n}\frac{1}{i} \quad (3.7)$$

On the other side the average number of necessary pages is:

$$P_N = \frac{N}{2(H_{B+1}-1)} \quad (3.8)$$

3.2.3 Active pages method

Active pages method is a compromise between the two previous methods. The sequential method always uses only one *Active Page*, the grouped method can use as many as it needs. The active pages method uses a fixed number of active pages q and uses the grouped method uses the grouped method if it doesn't exceed the number of alowed active pages q. Otherwise it proceeds as the sequential method would do.

Definition 3.8 (Active page)**.** *An active page is a page allocated in memory but not full.*

To sum up, the Active pages method works as follows.

- It starts by initializing q empty pages;
- if the parent of a node a is in a active page a is added in that page, otherwise the node is added to the active page with the least number of nodes (the one having more space left);

- if after adding the node a page is full a new one is allocated in mempry.

The number of pages allocated by the Active pages method is at most q plus the number of full pages P. The average number of accesses to pages decreses as q increases, going from the value of sequential method $q = 1$ to the value of the grouped method $q \to \infty$.
The active pages method with $q = 1$ is the sequential method, with $q \to \infty$ is the grouped method.

Exercise example 3.3. *Build a binary tree with the following keys: 49, 58, 73, 10, 62, 42, 40, 25, 64, 45, 72, 28, 77, 32, 9, 34, 76, 33, 81, 18 and store it in pages of* $B = 4$ *nodes using the Active Method with* $q = 2$.

Chapter 4

B-Trees

Regardless of the pagination method used, some pages have a much higher access frequency than others. For example, the first page containing the root must always be read during every search. In sequential methods, even the second page is frequently accessed. Inevitably, these frequently used pages will be removed from the first-level memory when searching a very large tree that requires accessing more than M pages, whether using the **LRU (Least Recently Used)** or **FIFO (First In, First Out)** replacement strategy. To optimize performance, the first page could be excluded from replacement, ensuring it remains in the first-level memory and reducing the number of I/O operations per search.
In general, binary trees work well, but in certain cases, their performance drops significantly. For example, a highly unbalanced tree can drastically increase the number of I/O operations, making it approach the $O(N/B)$ complexity of sequential search. This highlights the need to balance the structure, just as in main memory.
AVL trees are not well suited for pagination because rotations required for balancing might involve nodes in different pages, leading to excessive page swaps due to pointer modifications. Since AVL trees are inefficient for this scenario, **2-3 trees** become a better alternative. In fact, **B-trees**, which are derived from 2-3 trees, are the ideal structure for external memory.

Definition 4.1 (B-Tree)**.** *Let t be a fixed integer with $t \geq 2$, called the minimum degree. A non-empty **B-tree** of minimum degree t for a collection of records $\mathcal{C}$ with key K that does not allow duplicates is a tree with a root such that:*

1. *All leaves are at the same level.*
2. *Every node v (except for the root) contains $c(v)$ ordered keys $k_1 < k_2 < \cdots < k_{c(v)}$ with*
$$t - 1 \leq c(v) \leq 2t - 1. \tag{4.1}$$
3. *The root has at least one key and at most $2t - 1$ ordered keys.*
4. *Every node v has $c(v) + 1$ children.*
5. *The keys $k_1, \ldots, k_{c(v)}$ of each node v separate the key intervals stored in each subtree.*

An analogy with 2-3 Trees is that B-Trees can be seen as (2t-1)-(2t) Trees. An example of a B-Tree with $t = 3$ is shown in Figure 4.1. The B-Tree is a balanced tree that allows for efficient searching, insertion, and deletion operations.

Proposition 4.1. *The height h of a B-Tree with N keys satisfies*
$$h \leq \log_t \frac{N+1}{2} \tag{4.2}$$

Proof. The root node contains at least one key and every other node contains at least $t - 1$ keys. When maximum height is considered, the number of elements in each node is minimized, thus the root as degree 2 and every other node has degree t.
Because of that we will have $2t^{i-1}$ nodes at level i so the number of keys in the tree is:
$$N \geq 1 + (t-1)\sum_{i=1}^{h} 2t^{i-1} = 1 + (t-1)2\frac{t^h - 1}{t-1} = 2t^h - 1 \tag{4.3}$$

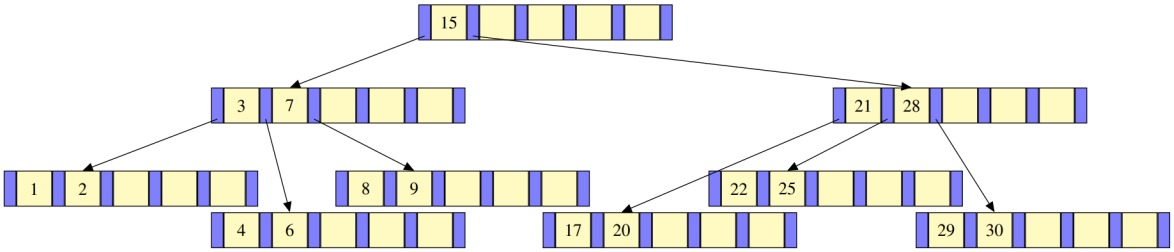

Figure 4.1: B-Tree with $t = 3$, $N = 17$, $h = 2$

Which implies $t^h \leq \frac{N+1}{2}$ and taking the logarithm we get:

$$h \leq \log_t \frac{N+1}{2} \tag{4.4}$$

□

Theorem 4.1. *The number of keys N in a B-Tree of height h and minimum degree t satisfies the following inequality:*

$$2t^h - 1 \leq N \leq (2t)^{h+1} - 1 \tag{4.5}$$

Proof. The first inequality is already demonstrated in the previous proposition (Equation 4.3). For the second inequality we can use the fact that the maximum number of keys in a B-Tree is obtained when every node has $2t - 1$ keys. In this case, the number of keys in each node is maximized, and we have:

$$N \leq (2t-1)\sum_{i=0}^{h}(2t)^i = (2t-1)\frac{(2t)^{h+1}-1}{(2t)-1} = (2t)^{h+1} - 1 \tag{4.6}$$

□

Exercise example 4.1. *How many elements can be stored in a B-Tree of height $h = 3$ and minimum degree $t = 100$? (Give a range for N using the previous theorem).*

Definition 4.2 (Loading factor)**.** *The loading factor α of a B-tree is the ratio between the number of keys present in the tree and the total number of keys that could be allocated in it, i.e., $(2t-1)g$, where g is the number of nodes in the tree.*

Proposition 4.2. *The average loading factor of a B-tree is approximately $\alpha \approx 0.7$.*

4.1 Searching on B-Trees

Let r be the root of the tree, k the key to search. The algorithm works as follows:

Search(**key** k, **root** v)

```
1: i ← 0
2: while i < c(v) and k > k_{i+1} do
3:     i ← i + 1
4: end while
5: if i < c(v) and k_{i+1} = k then
6:     return elem_i
7: else
8:     if v is a leaf then
9:         return null
10:    else
11:        return SEARCH(i-th child of v, k)
12:    end if
13: end if
```

Since every recursive call goes to the next level of the tree the algorithm has $O(\log_t N)$ complexity. The biggest advantage is gained when each node of the B-Tree occupises exactly a page of the external memory. This happens when $t = cB, c \in \mathbb{N}$ and in this case the number of I/O operations corresponds to the height of the tree.

$$T(N, M, B) \in O(\log_t N) \approx O(\log_B N) \tag{4.7}$$

The number of elementar operations is influenced by the fact that in each node we need to look for the key k, since the keys are sorted this has $O(log_2 t)$ complexity. The total work complexity is:

$$W(N, M, B) \in O(\log_t N \cdot \log_2 t) \approx O(\log_B N \cdot \log_2 B) = O\left(\frac{\log_2 N}{\log_2 B} \cdot log_2 B\right) = O(\log_2 N) \tag{4.8}$$

4.2 Insertion on B-Trees

The instertion of an element e with key k is performed by searching k in the tree and inserting k in the right leaf f. The research has to fail because in B-Trees every key must be unique. The algorithm works as follows:

- if the leaf f is not full, insert k in f and stop;
- if the leaf f is full, **split** it into two leaves f_1 and f_2 and insert the median key in the parent node of f and set the two pointers to refer to f_1 and f_2. Since each leaf contains at most $2t-1$ elements when it's full the median key is the t-th one. If the parent node is also full the spilit is repeated recursively until the root is reached. If the root is split a new root is created and the height of the tree increases by one.

The split operation works as follows:

- create a new node g and copy the last t keys from f to g;
- leave the first $t-1$ keys in f;
- put the t-th key in the parent node of f and set the two pointers to refer to f and g;

The split has work complexity $W(N, M, B) = O(t) \approx O(B)$ e transfer complexity $T(N, M, B) = O(1)$. The insert requires, in the worst case scenario, to perform the split operation h times, where h is the height of the tree. This brings the total complexity of the insertion to:

$$W(N, M, B) = O(t\ log_t N) \tag{4.9}$$

$$T(W, N, B) = O(log_t N) \tag{4.10}$$

4.3 Deletion from B-Trees

The deletion of a key k from a B-Tree is performed by searching for k in the tree. The algorithm works as follows. If k is **not in a leaf node**, replace k with his successor k' recursively and delete k' from the leaf node.

Definition 4.3 (Almost-empty)**.** *A leaf f is almost empty if it contains at least t keys.*

If k is in a leaf node f and f is **not almost empty** we can simply delete k from f. If f is almost empty we have to do the following:

- if the left or the right sibling of f is not almost empty we redistribute the keys between f and the sibling. We will also need to modify the separating key in the parent node;
- if both siblings are almost empty we merge f with one of the siblings (this operation is called **fuse**). To do so we add to one of the sibling the keys of f and the separating key in the parent node. The parent node will lose one key and one pointer. If the parent node is almost empty we repeat the operation recursively until we reach the root. If the root becomes empty we remove it and decrease the height of the tree by one.

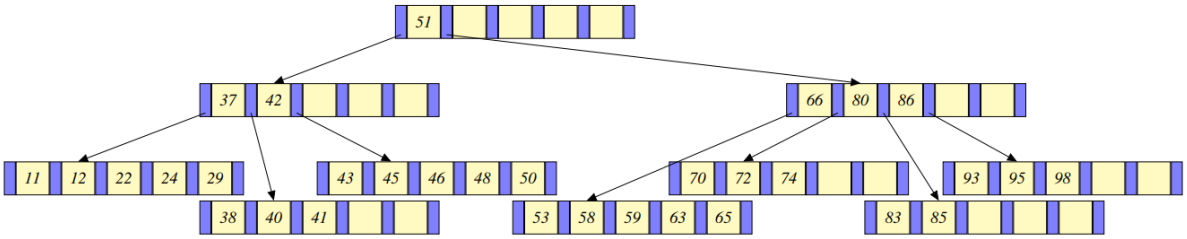

Figure 4.2: Solution to Exercise 4.2

So in the worst case scenario the delete will have to perform the fuse operation h times, where h is the height of the tree. The work complexity and the transfer of the delete operation are:

$$W(N, M, B) = O(t\ log_t N) \tag{4.11}$$

$$T(W, N, B) = O(log_t N) \tag{4.12}$$

Exercise example 4.2. *Considering a B-Tree of minimum degree $t = 3$, insert the following keys: 11, 29, 40, 37, 66, 51, 46, 80, 24, 12, 53, 98, 85, 59, 38, 45, 22, 65, 42, 72, 93, 43, 70, 83, 58, 95, 74, 48, 41, 50, 63, 86. A solution is provided in Figure 4.2.*

Chapter 5

B+ Trees

In **B-trees**, leaf node pointers are null, wasting space. To optimize this, we can **eliminate pointers in leaves** and store more keys, increasing storage efficiency without changing the tree height.

Similarly, increasing space in **internal nodes** allows for more pointers, leading to **fewer levels** and a shorter tree height. Since full records are usually stored in B-trees, keeping them only in the **leaves** while storing only **keys in internal nodes** further reduces tree height.

Additionally, **linking leaves in a bidirectional chain** optimizes **range queries**, as elements remain sorted. This structure is called a **B+ tree**, an improved version of the B-tree, ideal for **primary indexing** without duplicate keys.

Definition 5.1 (B+ Tree)**.** *A **B+ tree** of minimum degree t for a collection of records C with key k, which does not allow duplicates, is a search tree with the following properties:*

1. *Each node occupies a block.*
2. *All leaves are at the same depth and are linked by a bidirectional chain.*
3. *A non-root internal node contains $\delta \in [t-1, 2t-1]$ distinct keys, $\delta+1$ pointers to children, and a pointer to its parent.*
4. *A non-root leaf contains $\beta \in [b, 2b]$ records ordered by key, two pointers to the previous and next leaves in the chain, and a pointer to its parent.*
5. *The root, if it is not a leaf, contains $\delta \in [1, 2t-1]$ keys and $\delta+1$ pointers to children. If it is a leaf, it contains $\beta \in [0, 2b]$ records ordered by key.*
6. *The leaves, read from left to right, contain all records of C sorted by key.*
7. *Given an internal node v with keys $k_1 < k_2 < \cdots < k_\delta$ and children $w_1, w_2, \ldots, w_{\delta+1}$, the following holds:*
   - *The subtree rooted at w_1 contains a subset of records with keys smaller than k_1.*
   - *The subtree rooted at w_i, for $1 < i \leq \delta$, contains records with keys in the interval $[k_{i-1}, k_i)$.*
   - *The subtree rooted at $w_{\delta+1}$ contains records with keys greater than or equal to k_δ.*

The parameters t and b are defined as follows:

- *t is the largest positive integer such that a block can contain $2t-1$ keys and $2t+1$ pointers (including the pointer to the parent node).*
- *b is the largest positive integer such that a block can contain $2b$ records and 3 pointers to other blocks (parent, left sibling, and right sibling).*

From this definition, it follows that the same key cannot appear more than once in internal nodes.

Proposition 5.1. *A B+ tree with parameters t, b of height h has a number of elements N that satisfies*

$$2t^{h-1}b \leq N \leq (2t)^h 2b \tag{5.1}$$

Theorem 5.1. *A B+ tree with parameters t, b containing N records has height h such that*

$$h \in \Theta\left(1 + \log_t \left\lceil \frac{N}{b} \right\rceil\right) \tag{5.2}$$

Proof. The proof is similar to the one for B-trees. Let h be the height of the tree (thus, the levels range from 0 to h, making a total of $h+1$ levels). If $N \leq 2b$, there is only the root node, resulting in a single level.
Now, suppose $N > 2b$. Since the root has at least 2 children and each internal node has at least t children, at level i of the tree, there are at least $2t^{i-1}$ nodes for $1 \leq i \leq h$. Thus, the tree has at least $2t^{h-1}$ leaves. Since each leaf contains at least b records, we obtain:

$$N \geq 2t^{h-1} \cdot b$$

which leads to:

$$h - 1 \leq \log_t \frac{N}{2b}$$

On the other hand, every internal node, including the root, has at most $2t$ children, so the maximum number of leaves is $(2t)^h$ Since each leaf contains at most $2b$ records, we get:

$$N \leq (2t)^h \cdot 2b$$

which simplifies to:

$$h \geq \log_{2t} \frac{N}{2b}$$

thus, we finally obtain:

$$\frac{\log_t \frac{N}{b} - \log_t 2}{\log_t 2t} \leq h \leq 1 + \log_t \frac{N}{b} - \log_t 2$$

Since all terms except $\log_t \frac{N}{b}$ are constants, the thesis follows. □

Exercise example 5.1. *How many elements can be stored in a B+ tree of height $h = 3$ and parameters $t = 3, b = 2$? (Give a range for N using Preposition 5.1).*

5.1 Searching on B+ Trees

Searching on B+ Trees is similar to searching on B-Trees. The only difference is that in B+ Trees, the search is performed only in the leaves.
Given a key value k, the goal is to find the leaf (if it exists) that contains the record with key k. The method involves traversing the tree, following for each node the subtree w_i such that:

$$k_{i-1} \leq k < k_i$$

where we assume $k_0 = -\infty$ and $k_{\delta+1} = +\infty$, for $i = 1, \ldots, \delta + 1$. Upon reaching a leaf u, we search for k among its records. If k is found, the associated record is returned; otherwise, the search ends unsuccessfully.
Since each node is stored in a single page, the number of I/O operations corresponds to the number of levels in the tree:

$$T(N, t, b) \in \Theta\left(1 + \log_t \left\lceil \frac{N}{b} \right\rceil\right) \tag{5.3}$$

The I/O complexity decreases as t and b increase, meaning larger page sizes improve performance.
Regarding the number of elementary operations, searching within a page can be done using *binary search*, which requires at most $\log_2(2t)$ operations in internal nodes and $\log_2(2b)$ operations in leaves. Since there are h levels of internal nodes, we obtain:

$$W(N, t, b) \in O\left(\log_2(2t) \cdot \log_t \left\lceil \frac{N}{b} \right\rceil + \log_2(2b)\right) \tag{5.4}$$

which simplifies to:

$$O\left(\log_2(2t) \cdot \frac{\log_2 \left\lceil \frac{N}{b} \right\rceil}{\log_2 t} + \log_2(2b)\right) = O\left(\log_2 \left\lceil \frac{N}{b} \right\rceil + \log_2 b\right) \tag{5.5}$$

5.2 Insertion on B+ Trees

The insertion of a record r with key k involves an unsuccessful search that identifies the leaf u where r should be inserted. If u contains **fewer** than $2b$ records, r is inserted in order among the existing records. However, if u already contains $2b$ records, a *leaf split* is required. Let $r_1, \ldots, r_{2b+1}$ be the ordered records in u, including r. The node u is split into two nodes u_1 and u_2:

- u_1 contains records $r_1, \ldots, r_b$
- u_2 contains records $r_{b+1}, \ldots, r_{2b+1}$
- The key k_{b+1} (median value) of record r_{b+1} is inserted into the parent of u with pointers to u_1 and u_2.

If u was the *root*, a new root node is created containing k_{b+1} and pointers to u_1 and u_2. If the *parent of* u was already full (i.e., contained $2t-1$ keys), it must also be split like we saw for B-Trees. This splitting process may propagate up to the root, so the number of I/O operations is proportional to the height of the tree:

$$T(N,t,b) \in \Theta\left(1 + \log_t \left\lceil \frac{N}{b} \right\rceil\right) \tag{5.6}$$

Regarding *work complexity*, each node split requires moving *half of its data*, i.e.,

- $O(t)$ operations for *internal nodes*
- $O(b)$ operations for *leaves*

Thus, the total work complexity is:

$$W(N,t,b) \in O\left(b + t \cdot h\right) = O\left(b + t \log_t \left\lceil \frac{N}{b} \right\rceil\right) \tag{5.7}$$

5.3 Deletion on B+ Trees

The deletion of a record r with key k requires first locating the leaf u where r is stored. The record r is then removed. If u is the **root** or still contains more than b records after deletion, the process terminates. Otherwise there would be only b-1 records in u, let $r_1, \ldots, r_{b-1}$ be the remaining records in u.
If the **right sibling** (or left sibling) of u, denoted as u', contains more than b records, a **redistribution** is performed. Let v be the parent of u, and let $\hat{k}$ be the key that separates u and u'. Records are redistributed between u and u', and $\hat{k}$ in v is updated accordingly.
If u' has exactly b records, a **merge** is performed. All records from u' are moved into u, and the separator key $\hat{k}$ is removed from v. If v still contains at least $t-1$ keys or is the root, the process stops. Otherwise, if v now has fewer than $t-1$ keys, the procedure recurses upwards.
The number of I/O operations required is proportional to the height of the tree:

$$T(N,t,b) \in \Theta\left(1 + \log_t \left\lceil \frac{N}{b} \right\rceil\right) \tag{5.8}$$

Since each merge or redistribution involves shifting data, the **work complexity** is determined by:

- $O(t)$ operations for *internal nodes.*
- $O(b)$ operations for *leaves.*

Thus, the total complexity is:

$$W(N,t,b) \in O\left(b + t \log_t \left\lceil \frac{N}{b} \right\rceil\right) \tag{5.9}$$

5.4 Range query on B+ Trees

The search for all records with keys in the interval $[k_1, k_2]$ begins by locating the leaf containing a record with key k_1. Once found, the search continues to the right, scanning successive leaves to collect all records with keys within $[k_1, k_2]$. Let R be the number of records retrieved. Since the leaves are linked by a bidirectional chain, scanning successive leaves is very efficient. Let r be a record in a leaf f, the operation **next**(r) is performed in $O(1)$ time. If the r is the rightmost element in f is successor is the first element of the next leaf otherwise is the next element in the same leaf.
The I/O cost consists of the search operation plus accessing the subsequent leaves, yielding:

$$T(N, t, b) \in \Theta\left(1 + \log_t \left\lceil \frac{N}{b} \right\rceil + \left\lceil \frac{R}{b} \right\rceil\right) \tag{5.10}$$

For the work complexity, the search requires:

$$W(N, t, b) \in O\left(\log_t \left\lceil \frac{N}{b} \right\rceil + \log b + R\right) \tag{5.11}$$

Exercise example 5.2. *Insert the following keys into a B+ tree with parameters $t = 3$ and $b = 2$: 55, 65, 81, 11, 69, 47, 45, 27, 71, 49, 80, 31, 86, 36, 10, 38, 84, 37, 89, 20.*

Chapter 6

Hash tables

In general techniques based on comparing keys have at best $O(\log n)$ complexity, with n beeing the total number of keys. There exist search techniques that do not rely on comparisons and can achieve **constant-time** complexity, meaning the time required is independent of the number of data items. These methods are based on transforming keys into addresses.

If we had distinct integer keys ranging from 1 to N, we could store the data with key i in the i-th position of a table. In this way, data access would be immediate.

Hashing methods generalize this very particular situation to cases in which the keys do not have such ideal properties. The first step of a hash-based search consists of computing a *hash function* that transforms the search key into an address within the table where data is stored. That is, a function

$$h : K \to \{0, 1, \ldots, m-1\}$$

where K is the set of keys and $\{0, 1, \ldots, m-1\}$ are the addresses of the table positions.

Ideally, different keys $k_1 \neq k_2$ should map to different addresses $h(k_1) \neq h(k_2)$, but unfortunately, hash functions are rarely injective. As a result, two or more distinct keys may produce the same address.

When $h(k_1) = h(k_2)$ for $k_1 \neq k_2$, a **collision** is said to occur, and k_1 and k_2 are called **synonyms**. The second part of a hash-based search concerns the mechanisms used for **collision resolution**.

Searching in a hash table

Suppose we have a hash function h and a collision resolution method M. The search for a key k proceeds as follows:

1. Compute $h(k)$ and check the element at that position.
2. If the element has key k, the search concludes successfully. Otherwise, it is still unclear whether k is absent from the table or whether a collision occurred during insertion and k was stored at a different position.
3. Apply the method M to k; either the key is found or it is concluded that k is not present.

If h is a good hash function and produces few collisions, then most data can be accessed with a single step and comparison. In the case of collisions, if M is a good resolution method, the number of accesses is very limited, and thus the overall search time remains nearly constant.

Hash function

A hash function must transform keys (typically integers or strings) into integers in the range $[0, m-1]$, where m is the number of elements in the table. A good hash function h should satisfy the following properties:

1. It should be easy to compute, so as not to burden the auxiliary operations related to the search;
2. It should generate values that are uniformly distributed over the interval $[0, \ldots, m-1]$.

Definition 6.1 (Perfect hash function)**.** *A hash function is said to be* ***perfect*** *if it is injective, meaning that it maps distinct keys to distinct addresses. In this case, the search time is constant and equal to one access.*

In general, it is not possible to construct a perfect hash function for all possible keys, this would require a very high value of m. The simplest hash function for numeric values is

$$h(k) = k \mod m \tag{6.1}$$

To deal with alphabetic strings we need to enstablish a rule to convert them into integers. For example we can consider alphabetic characters as numbers in base 26. In this case we define the hash function as follows. Let k be a string of length n, k_i be the i-th character of the string, $v(k_i)$ the corrisponding value.

$$h(k) \doteq \sum_{i=0}^{n-1} v(k_i) \cdot 26^i \mod m \tag{6.2}$$

The choice of m is very important. If m is a prime number, the distribution of the hash function is more uniform. In general, it is advisable to choose m as a prime number. This is due to the properties of the module operation.

Another widely used method for alphabetic keys is the **folding method**. Suppose the key is a string of 12 characters, each represented with 8 bits. We divide the key into 4 groups of 3 characters each and compute the *exclusive OR (XOR)* of the four groups.
A property of XOR is that each input influences the final result, which in this case is a 24-bit value that depends on all 12 characters of the original string. These 24 bits can be interpreted as a number, to which we can apply, for example, the modulo operation to obtain the final value in the range $[0, \ldots, m-1]$.

6.1 Collision resolution methods

Collesion resolution methods can be divided into two categories, those that use chained structures to memoraize synonyms and those that recompute new addresses in case of collision.

Separate chaining methods

The collision resolution method known as *separate chaining* organizes the table such that its elements contain only pointers to chains. Each chain stores all the data items that are *synonyms*, i.e., keys for which the hash function computed the same address — the address corresponding to the table position containing a pointer to the head of the chain.

To insert a new key k, one computes $h(k)$ and inserts k into the chain pointed to by the entry at position $h(k)$ of the table. Searching is done in a similar way: the value of the hash function determines the chain to access, which is then traversed sequentially.

Note that this is the only case of a hash table in which it is possible to store a number of elements n greater than the length m of the table.

Exercise example 6.1. *Use the separate chaining method to store the following keys in a hash table of size $m = 11$ using hash function $h(k) = k \mod m$. $1, 19, 5, 12, 18, 3, 8, 9, 14, 7, 16, 24, 23, 13, 27, 34, 38$. The solution is shown in Figure 6.1.*

Joined chaining methods

A structurally simpler method, though limited by the condition $n \leq m$—is the implementation of hash tables using *coalesced chaining*. In this case, each element in the table consists of two fields: one containing the key and the other an index.

Each element is inserted into the position indicated by the hash function. If that position is free, the key is placed there and the index is set to -1. When a collision occurs, the new key k is inserted into the first available position after $h(k)$ (the table is considered circular), and the index of position $h(k)$ is updated to point to this new location.

If another synonym is inserted later, it first accesses $h(k)$ and finds it occupied. It then follows the index to the position of the first synonym and proceeds in the same way—being placed in the first free position after that, with its index linked from the previous entry.

It may happen that, when inserting a key k_1, the position $h(k_1)$ is already occupied by a key k_2 that is *not* a synonym but ended up there while searching for a free position. In this case, it is preferable to replace k_2 with k_1 (so k_1 can be accessed in a single step) and reallocate k_2. This operation does not increase the number of steps required to retrieve k_2.

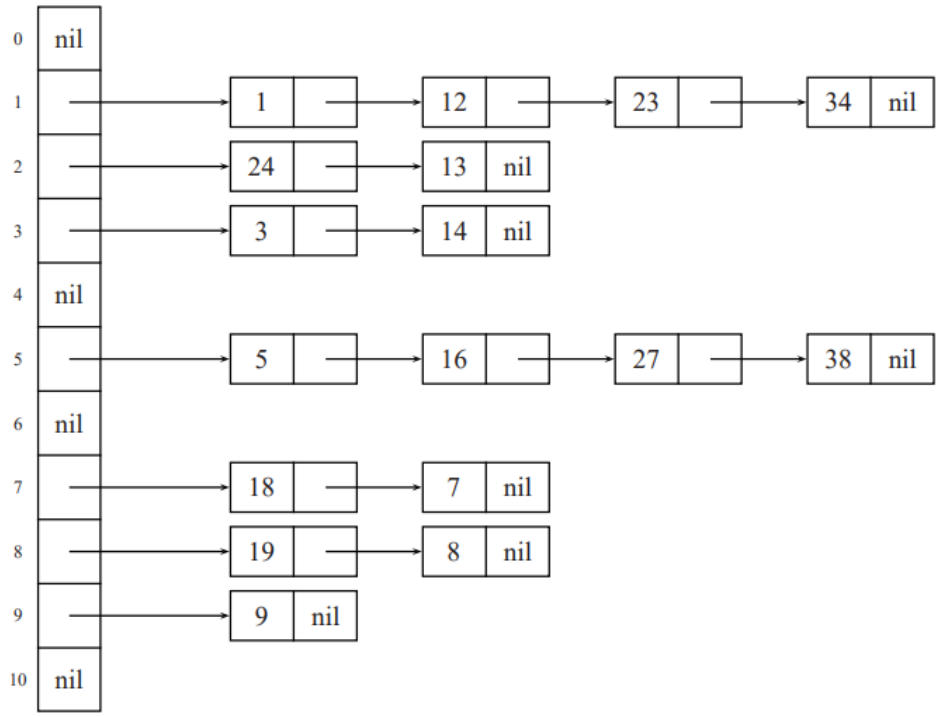

Figure 6.1: Hash table with separate chaining method

Exercise example 6.2. *Use the joined chaining method to store the following keys in a hash table of size* $m = 11$ *using hash function* $h(k) = k \mod m$. $1, 19, 5, 12, 18, 3, 8, 23, 14, 7, 17$. *The solution is shown in Figure 6.2.*

Open addressing method

Hash methods that use chaining require additional memory to store pointers or indices. By observing the coalesced hashing method, one might consider removing the indices and instead performing a *sequential search* for synonyms starting from the position $h(k)$.

This approach saves space but increases the number of comparisons required during search. An unsuccessful search ends when an empty position is encountered, so to avoid infinite loops, it is necessary to ensure that $n < m$. Moreover, the greater the number of empty slots, the faster the search can

	chiave	indice
0	14	-1
1	1	2
2	12	4
3	3	0
4	23	-1
5	5	-1
6	17	-1
7	18	10
8	19	9
9	8	-1
10	7	-1

Figure 6.2: Hash table with joined chaining method

conclude. The cost of search depends in fact on the **load factor** of the hash table.

$$\alpha = \frac{n}{m} \tag{6.3}$$

The insertion with open addressing is performed as follows. Suppose that empty positions are marked with $T[x] = -1$.

Insertion (**key** k, **table** T)

```
1: x = h(k)
2: while T[x]! = −1 do
3:     x = (x + 1)  mod m
4: end while
5:  T[x] = k
```

Exercise example 6.3. *Use the open addressing method to store the following keys in a hash table of size* $m = 19$ *using hash function* $h(k) = k \mod m$. $1, 19, 5, 20, 18, 3, 8, 9, 14, 7, 24, 43, 39, 13, 16, 12, 62$.

Theorem 6.1. *When the collision resolution method is open addressing, the average number of comparisons required to search for a key* k *in a hash table of size* m *with* n *is*

$$\frac{1}{2}\left(1 + \frac{1}{1-\alpha}\right) \tag{6.4}$$

in case of success and

$$\frac{1}{2}\left(1 + \frac{1}{(1-\alpha)^2}\right) \tag{6.5}$$

in case of failure.

6.2 Hash in external memory

Just like tree structures, hashing techniques can also be adapted to secondary memory and retain the same advantages and limitations observed in main memory:

1. Random access **performance is better** than with tree-based structures.
2. Key dispersion makes **any sorting operation extremely costly**.
3. Due to the scattering of keys, range queries are not possible.

However, when neither sorting nor range queries are required, hash-based organization is highly efficient, typically requiring just over one access to secondary memory per search. This makes hashing very practical in real-world applications such as record storage and index management.

From a technical perspective, the main drawback of hash-based organization is its **static nature**: while tree structures can grow by adding pages as the number of keys increases, hash techniques require that the archive size be known in advance. Expanding it later involves an expensive full rehashing of the entire archive. For this reason, the most significant research on hashing in *External Memory (EM)* has focused on techniques for making hash structures dynamic.

6.2.1 Static hashing

The idea behind static hashing is to use a hash function that maps the keys into the pages (**buckets**) in witch they will be stored. Untill the page is full synonyms are added sequantially in the same page, the collision resolution problem occurs when the page is full.

Let B be the number of keys that can be stored in a page, N the total number of keys. To solve the collisions there are two different approaches:

- collect all the keys that cannot be stored in the designed page in **overflow pages**;
- similar to the *separate chaining method* , when a page of synonyms is full store the new synonyms in a new page linked to the previous one with a linked list.

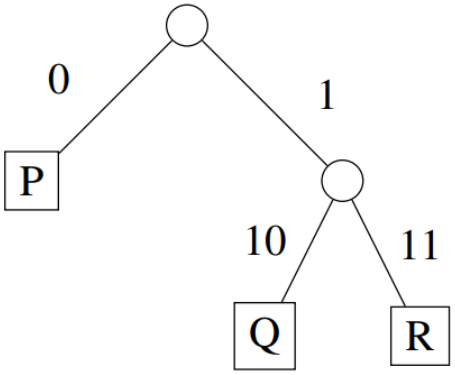

Figure 6.3: Dynamic hashing

Like in the case of main memory, the cost of searching depends on the load factor α of the hash table.

To memorise the keys we need a table contaning more than $\lceil \frac{N}{B} \rceil$ pages. In particular to have a load factor $\alpha = 0.8$ we need at least $\lceil 1.25 \frac{N}{B} \rceil$ pages.

The number of I/O operations required to search for a key k in a hash table of size m with n keys is, in the worst case, $O(\lceil \frac{N}{B} \rceil)$. Although, when the hash function distributes the keys uniformly the average number of I/O operations is:

$$\frac{\lceil \frac{N}{B} \rceil}{m} = \frac{\lceil \frac{N}{B} \rceil}{1.25 \lceil \frac{N}{B} \rceil} = \frac{1}{1.25} \tag{6.6}$$

6.2.2 Dynamic hashing

To enable an archive to grow dynamically while using a hash-based organization, the technique of *Dynamic Hashing* has been introduced. Suppose we have a pseudorandom function:

$$\sigma : K \to [0, m-1] \tag{6.7}$$

which transforms keys into binary-coded integers. For any key $k \in K$, let:

$$\sigma(k) = b_1^{(k)} b_2^{(k)} \dots b_{\lceil \log_2 m \rceil}^{(k)} \tag{6.8}$$

be the binary representation of the value obtained from σ. The insertion of keys proceeds as follows:

- Initially, a single page P is allocated, and all keys are inserted into it.
- When P becomes full, a second page Q is created.
- The keys in P are redistributed based on the first bit of their hash value:
  - If $b_1^{(k)} = 0$, the key remains in P.
  - If $b_1^{(k)} = 1$, the key is moved to Q.
- The new key that caused the split is inserted into P or Q, according to its first bit.

If σ is a good pseudorandom function, the keys will be distributed uniformly between the two pages.

When a page becomes full again, the process is repeated, but this time the second bit of the hash value is used to determine the destination page. This process continues until all bits of the hash value are used. Internal nodes do not contain keys but only two pointers to the pages corresponding to the two possible values of the next bit. This allows the tree to be contained in central memory, while the pages can be stored on disk. Since the entire tree can be stored in internal memory we will always need just one I/O operation to reach every key in the archive.

6.2.3 Extendible hashing, methods with directory

The idea behind Extendible hashing is to use multiway trees instead of binary trees. This will result in a lower load factor.

To overcome the issue of underloaded pages in multi-way tries, in 1978, Fagin, Nievergelt, Pippenger, and Strong proposed a new method called *Extendible Hashing*, which (almost always) requires only one or two I/O operations for search and insertion. Suppose we have a hash function

$$h : K \to \{0, 1, 2, \ldots, m-1\} \tag{6.9}$$

that maps keys to binary-encoded integers, where the range m of the hash function is chosen to be sufficiently large.

The method uses a **directory**, i.e., an array of 2^d pointers to pages. Each key k is assigned to the page pointed to by the directory location corresponding to the d most significant bits of $h(k)$. The value d is called the **Global Depth** and is initially set to the smallest value such that each page can contain at most B keys. The value d can be increased when that condition is no longer satisfied.

Two or more directory locations can point to the same page if the total number of keys assigned to them is less than or equal to B. In that case, such locations share the same page if they have the first c bits of their address in common, with

$$0 \leq c \leq d \tag{6.10}$$

The value c is called the **Local Depth** and represents the smallest number of bits needed to distinguish a group of keys in the same page.

Definition 6.2. *An extendible hash table of order d is a directory with 2^d references to pages, each of which can hold up to B records. All records in the same page P have the first c bits of $h(k)$ in common. The directory contains 2^{d-c} pointers to page P, starting from the location specified by those c leading bits.*

To maintain these properties, two key operations are required:

- *Page Split*: When a page overflows, it is split into two pages. The keys are redistributed between the old and new page according to the $(c+1)$-th bit, and the local depth is updated as $c \leftarrow c+1$.
- *Directory Doubling*: If after a split we have $c = d$, it becomes necessary to double the size of the directory by increasing d by one, thus expanding from 2^d to 2^{d+1} references to pages.

It is important to note that no page of the archive is modified during the doubling of the directory, except for the one that caused the overflow.

The deletion of an element can be performed in a similar way. When two blocks with the same local depth c contain elements whose hash addresses share the same $c-1$ most significant bits and their combined number of elements can fit into a single block, then their elements are merged into a single block with local depth $c-1$. The total size of the two blocks being merged must be sufficiently less than B in order to avoid an immediate need for a split upon subsequent insertions.

The directory is halved (and d is decremented by 1) when all local depths are less than the current value of d.

Proposition 6.1. *The extendible hash table, consisting of a set of keys, depends only on the values of the keys and not on the order in which the keys are inserted.*

Theorem 6.2. *The expected number of pages required to store n elements is asymptotically equal to*

$$\frac{n}{\ln(2)} \approx \frac{n}{0.69}. \tag{6.11}$$

Since $n = \frac{N}{B}$ is the minimum number of pages needed to store N elements the load factor is approximately $\alpha = 0.69$ and the total number of pages is 1.44 times the minimum number of pages needed.

It's also important to note that the directory is an array of links. If it all fits in RAM then the cost of searching will be $T(N, M, B) = 1$, if not a second evel can be added bringing the cost to $T(N, M, B) = 2$.

Methods without directory

As we saw a disadvantage of the extendible hashing method is that the directory can be very large and if it doesn't fit in RAM a second I/O operation is needed to access it. To overcome this problem, we can use a method that doesn't require a directory. Two possible techniques to achieve this are **virtual hashing** and **linear hashing**.

6.2.4 Virtual hashing

Virtual hashing is based on the following theorem.

Theorem 6.3. *Given an integer k and an integer divisor d, if $k \bmod d = r$, then*

$$k \bmod 2d = r \quad or \quad r + d. \tag{6.12}$$

Let $h : K \to \{0, 1, 2, \ldots, m-1\}$ with $m >>$. We define

$$h_0 : \mathbb{N} \to \{0, 1, 2, \ldots, d-1\},\ h_0(k) = h(k) \mod d \tag{6.13}$$

where $d \in \mathbb{P}$ is a small prime number. Virtual hashing works as follows.

Initially keys are inserted in d pages of the archive using the function $h_0(k)$. When a page p overflows, a re-hashing of the keys is performed using the function

$$h_1(k) = h(k) \mod 2d \tag{6.14}$$

wich distributes the keys between the page p and the page $p + d$. The page $p + d$ is created only if it is needed. In this way the archive doubles its number of pages from d to $2d$.