Unit 9 Space and Time Tradeoffs

Structure:

9.1 Introduction

By now you must be familiar with different methods of problem solving using algorithms. This unit deals with space and time tradeoffs in algorithms. Algorithm analysis determines the amount of resources necessary to execute the algorithm. The vital resources are time and storage. Most algorithms are constructed to work with inputs of arbitrary length. Usually the efficiency of an algorithm is stated as a function relating the input length to the number of steps (time complexity) or storage locations (space complexity).

A balance has to be maintained in the space and time aspects of computing. Research in Computer Science these days is more towards achieving time efficiency.

With continuous reduction in prices, space availability is perhaps not a significant problem. But with respect to networking and robotics, however, the necessity of a balance becomes apparent. Memory has to be used conservatively.

Space-time or time-memory tradeoff in context with algorithms relates to the execution of an algorithm. The execution of an algorithm can be done in a short time by using more memory or the execution time becomes more when memory used is less. Therefore selecting one alternative over the other is the tradeoff here.

Many problems such as sorting or matrix-multiplication, have many choices of algorithms to use, of which some are extremely space-efficient and some extremely time-efficient. Space-time tradeoff proves that no algorithm exists where efficiency is achieved by small space and small time simultaneously.

This unit includes the various approaches such as sorting by counting, input enhancement in string matching, hashing, and B-Tree technique where time is an important factor and the result of computation is quick at the cost of space.

Objectives:

After studying this unit you should be able to:

- explain the importance of space-time tradeoff in programming
- the process of sorting by counting
- analyze the process of input enhancement in string matching
- define the role of hashing in space and time tradeoff
- explain B-Tree technique with respect to space and time tradeoff

9.2 Sorting

Input enhancement is based on preprocessing the instance to obtain additional information that can be used to solve the instance in less time. Sorting is an example of input enhancement that achieves time efficiency.

First, let us study distribution counting.

9.2.1 Distribution counting

Distribution counting is a sorting method that uses some associated information of the elements and places the elements in an array at their relative position. In this method elements are actually distributed in the array from 0^{th} to $(n-1)^{th}$ position. This method ensures that the elements do not get over written. The accumulated sum of frequencies also called as distribution in statistics is used to place the elements at proper positions. Therefore this method of sorting is called distribution counting method for sorting.

6	4	2	6	4	4	2

Figure 9.1: Array a[ ]

The figure 9.1 depicts the values of array a[ ]. Observing the array we find that the list contains only {2, 4, 6}. We will count the frequencies of each element and the distribution count. Distribution count value represents the last occurrence of corresponding element in the sorted array. These values are given in table 9.1.

Table 9.1: Distribution Values of Elements of Array a[ ]

Elements	2	4	6
Frequencies	2	3	2
Distribution values	2	5	7

We will have two arrays as shown in figure 9.2, one to store distribution values in an array called Dval [0..2] and another array to store the sorted list of elements called b[0…6] and then scan the unsorted list of elements from right to left. The element that is scanned first is 2 and its distribution value is 2.

array Dval[ ] **array b[ ]**

Figure 9.2: Sorted Element Stored in Array b[ ]

Decrement the distribution value of 2, that is Dval[0]=Dval[0]-1. After which move to left and scan the element at a[5]. So insert element 4 at 5-1 = 4 i.e b[4] position as shown in figure 9.3.

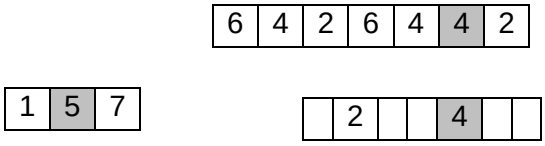


Figure 9.3: After Scanning Element a[5]

Now decrement the distribution value of 4 that is Dval 5-1 = 4. Scan a[4] which is 4 then find the distribution value of 4 Dval[1] = 4 put 4 at 4 - 1 = 3, at b[3] location. This is depicted in figure 9.4.

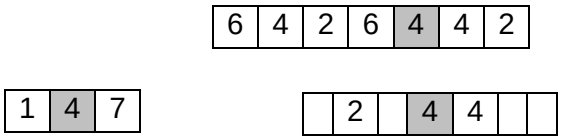


Figure 9.4: After Scanning Element a[4]

The array a[ ] is scanned through all the elements and sorted. In the last step, a[0] which is 6, is scanned. The distribution value of 6 is Dval[2] =6. Hence we insert 6 at 6-1=5 i.e. at b[5] position. This is shown in figure 9.5.

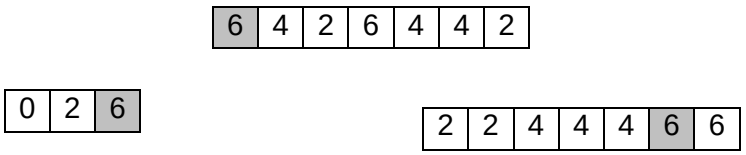


Figure 9.5: After Scanning All Elements in Array a[ ]

Thus the array is scanned from right to left. The following figure 9.6 depicts the summation of all the above steps.

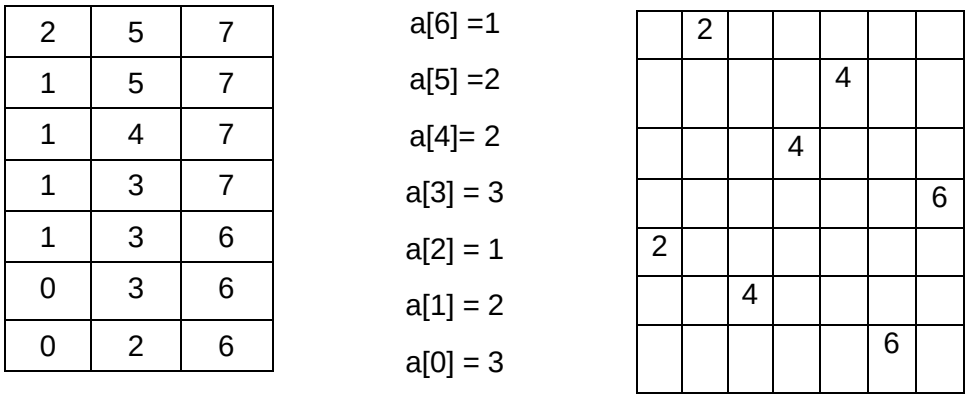


Figure 9.6: Depicting Array Dval[ ] and Array b[ ] After Scanning All Elements

The draw back of this method is that it depends on the nature of input, requires additional auxiliary array for sorting list and an array for distribution of values.

Let us now discuss the algorithm for distribution counting.

Algorithm for Distribution Counting

```
DistributionCounting(A[0...n-1],L,u)
//Sorts an array of integers from a limited range by distribution counting
//Input: An array A[0....n-1] of integers between L and u ( L<= u)
//Output: Array S[0...n-1] of A's elements sorted in nondecreasing order
for j = 0 to u-1 do D[]j] = 0 //initialize frequencies
for i = 0 to n-1 do D[A[i] - L] = D[A[i] -L] +1 //compute frequencies
for j =1 to u-L do D[j] = D[j-1] + D [j]
for 1=n - 1 downto 0 do
j = A[i] –l
S[D[j] - 1] = A[i]
D[j] = D[j] – 1
return S
```

Let us now trace this algorithm.

Algorithm Tracing for Distribution Counting

```
n=2
A[]=[1,2,1]
ALGORITHM DistributionCounting(A[0...1],2,3)
for j = 0 to 3 do D[0] = 0 //initialize frequencies
for i = 0 to 3 do D[1-2] = D[1-2] +1 //compute frequencies
for j =1 to 1 do D[1] = D[0] + D [1]
for 1=1 down to 0 do
j = A[0] –l=0
S[0 - 1] = 1 ie S[1]=1
D[1] = D[1] – 1
return 1
```

If we analyze the algorithm for distribution counting, we can divide the time taken into four parts as given in the table 9.2.

Table 9.2: Analysis of Distribution Counting

Loop	Complexity
for j = 0 to u-1 do D[]j] = 0	Θ(u)
for i = 0 to n-1 do D[A[i] - L] = D[A[i] -L] +1	Θ(n).
for j =1 to u-L do D[j] = D[j-1] + D [j]	Θ(u)
for 1=n - 1 downto 0 do j = A[i] –l S[D[j] - 1] = A[i] D[j] = D[j] – 1 ,it takes O(n)	Θ(n)
Total	Θ(u + n)

If u = O(n), then the time complexity of distribution counting will be Θ(n).

Self Assessment Questions

1. Input enhancement is based on _____ the instance.
2. The information which is used to place the elements at proper positions is accumulated sum of frequencies which is called as _____.
3. Sorting is an example of input enhancement that achieves _____.

9.3 Input Enhancement in String Matching

Now, let us see some algorithms for string matching.

String matching is an important application of input enhancement. We know that brute force method is the simplest method. But it is time consuming because every character of pattern is matched with every character of text. Hence faster algorithms have been developed and some of these are listed below:

- Horspool's algorithm
- The Boyer-Moore algorithm
- The Knuth-Morris-Pratt algorithm

Two string matching algorithms – the Boyer-Moore algorithm and Horspool's algorithm are discussed in this section. These algorithms are based on input enhancement and preprocess the input pattern and get some information about the input. They then arrange this information in a tabular form and then make use of the information in the actual algorithm.

9.3.1 Horspool's algorithm

How does the Horspool's algorithm work?

This algorithm places the pattern beside the text. The characters from the pattern are matched right to left with the characters from the text. If the character of the pattern matches the character in the text, then the desired substring is found as in the example shown in figure 9.7.

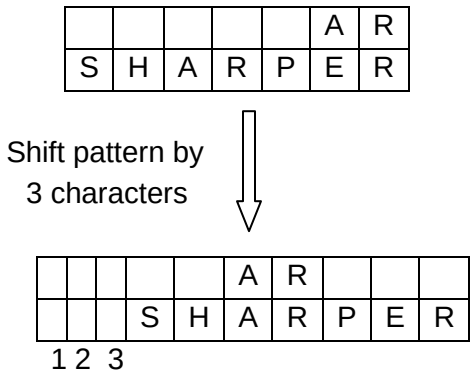


Figure 9.7: Example for Pattern Matching

The number of shifts that are made to match pattern against text is an important activity. To speed up the algorithm we can precompute the shift sizes and store them in a table. Such a table is called shift table.

The shift size can be computed by the following formula:

If 'T' is not among first m-1 characters of pattern

Table (T) = the patterns length m

Else Table (T) = the distance from the rightmost T among first m-1 characters of patterns to its last character.

The first step in this algorithm is to create the shift table for the pattern as given in table 9.3.

Initialize the shift table by length of pattern i.e. KIDS which is 4.

3	2	1	
K	I	D	S

fill ‘K’ column entry by 3,‘I’ column entry by 2 and D column entry by ‘1’

Table 9.3: Shift Table

A	B	C	D	E	F	G	H	I	J	K	L	M	N
4	4	4	1	4	4	4	4	2	4	3	4	4	4

O	P	Q	R	S	T	U	V	W	X	Y	Z
4	4	4	4	4	4	4	4	4	4	4	4

The table 9.4 illustrates the way the pattern is matched with the text.

Table 9.4: Matching Pattern Against Text

D A V I D _ L O V E S _ K I D S K I D S	Scan is done from Right to left I≠S, refer to the shift table I indicates 2, Hence shift pattern by 2 positions
D A V I D _ L O V E S _ K I D S _ _ K I D S _ _ _ _ _ _ K I D S _ _ _ _ _ _ _ _ _ _ K I D S	Mismatch occurs so the entire pattern is shifted to the right by its length 4
D A V I D _ L O V E S _ K I D S _ _ _ _ _ _ _ _ _ _ K I D S	Now I≠S, with reference to the shift table the pattern is shifted 2 positions to the right

<table><tr><td>D</td><td>A</td><td>V</td><td>I</td><td>D</td><td></td><td>L</td><td>O</td><td>V</td><td>E</td><td>S</td><td></td><td>K</td><td>I</td><td>D</td><td>S</td></tr><tr><td></td><td></td><td></td><td></td><td></td><td></td><td></td><td></td><td></td><td></td><td></td><td></td><td>K</td><td>I</td><td>D</td><td>S</td></tr></table>	All the characters from the text are matching the characters in the pattern, hence the search is declared successful

Let us next discuss the Horspool's algorithm for string matching.

Horspool Algorithm for Matching Strings

```
Horspool_string(P[ ],T[ ],m,n)

//Input: The pattern P[ ], the text T[ ], length of pattern m, length of text n
//Output: The first positional index of matching substring
//If no matching substring is found then return -1 shift_table(P[0..m-1])
//Construct the shift table i←m-1
//set the position from right end in the pattern
While(i<=n-1) do
{
  //initially assume that the number of matched characters are 0
  match_ch←0
  while((match_ch<=m-1) AND (P [m-1-match_ch] ))
  //Scan pattern from right to left to locate rightmost occurrence of
character
    match_ch←match_ch+1
  if (match_ch=m) then
    return i-m+1//shifts pattern to its right if only last character is matching
  else
    ii+Table[T[i] ]//refer to the characters entry in the shift table
}
Return -1
```

Let us now trace this algorithm.

Algorithm tracing for Horspool's Algorithm

```
n=2
P[ ]=[EE]
T[ ]=[GREEDY]
m=5
Horspool_string(P[ ],T[ ],m,n)
While(i<=1) do
{
  //initially assume that the number of matched characters are 0
  match_ch=0
  while((match_ch<=4) AND (5 ))
  //Scan pattern from right to left to locate rightmost occurrence of
character
    match_ch=0+1
  if (match_ch=5) then
    return 1-5+1=5//shifts pattern to its right if only last character is
matching
  else
    0+Table[5]//refer to the characters entry in the shift table
}
Return -1
```

Time and space complexity of Horspool's algorithm for string matching is given in two parts.

- Preprocessing phase time complexity – O(m+n) and space complexity – O(n)
- Searching phase time complexity – O(m + n)

The average number of comparisons for one text character is between 1/п and 2/(п+1) where п is the number of storing characters.

Next let us study the Boyer- Moore algorithm.

9.3.2 Boyer-Moore algorithm

The Boyer-Moore algorithm uses two heuristics: good-suffix and bad-character shift.

Bad character shift

We use this when mismatch occurs. We decide the number of places to shift by using bad character shift.

- As in Horspool's algorithm if the rightmost character does not match, then the pattern is shifted to the right by its length.
- When the rightmost character of the pattern matches with that of the text, then each character is compared from right to left. If at some point a mismatch occurs after a certain number of 'k' matches with text's character 'T', then bad character shift is denoted by P.

$$P = \max \{ \text{text (T)} - k, 1\}$$

Good suffix Shift

This shift helps in shifting a matched part of the pattern, and is denoted by Q. Good suffix shift Q is applied after $0 < k < m$ characters are matched.

Q = distance between matched suffix of size k and its rightmost occurrence in the pattern that is not preceded by the same character as the suffix

When the substring is not found do the following:

- If all the m characters of pattern are matching with the text then stop
- If a mismatching pair occurs, then compute bad character shift P and good suffix shift Q and use these to compute shift size.

$$R = \max \{P,Q\} \text{ where } k > 0$$

Consider

B	H	U	T	T	O		K	N	O	W	S		T	O	R	O	N	T	O
T	O	R	O	N	T	O													

6 5 4 3 2 1 0

T	O	R	O	N	T	O

The bad character shift table is as in table 9.5

Table 9.5 Bad Character Shift Table

A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R
7	7	7	7	7	7	7	7	7	7	7	7	7	2	3	7	7	4

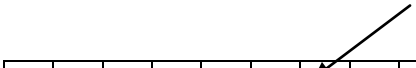
S	T	U	V	W	X	Y	Z
7	1	7	7	7	7	7	7

The Good- suffix table is as in Table 9.6

Table 9.6: Good-Suffix Table

k	Q
1	3
2	5
3	5
4	5
5	5
6	5

Space means shift by 7



B	H	U	T	T	O		K	N	O	W	S		T	O	R	O	N	T	O
T	O	R	O	N	T	O													

Since space is encountered while matching the pattern with the text, the pattern is shifted by 7 positions as shown in table 9.7.

Table 9.7: After Shifting the Pattern by 7 Positions

B	H	U	T	T	O		K	N	O	W	S		T	O	R	O	N	T	O
							T	O	R	O	N	T	O						

With reference to T entry from table 9.5 i.e. bad character table which is 1, shift the pattern to the right by 1 position as shown in table 9.8.

Table 9.8: Pattern Shifted to Right by 1 Position

B	H	U	T	T	O		K	N	O	W	S		T	O	R	O	N	T	O
								T	O	R	O	N	T	O					

k=2

There is mismatch in pattern because of the blank place in the text. Therefore P = max [(text (-) – k) ,1]

= max {(7 - 2),1}

P = 5

Compute Good suffix shift Q. As 2 characters from the pattern are matching Q(2) = 5

Shift size R = max {P,Q}
= max {5,5}
= 5

Therefore we shift the pattern 5 positions ahead as shown in table 9.9.

Table 9.9: Depicting the Pattern in the Text

B	H	U	T	T	O		K	N	O	W	S		T	O	R	O	N	T	O
													T	O	R	O	N	T	O

The required pattern is found in the text

Following are the steps that are followed to find the pattern:

Step 1: Construct a bad character table and obtain the bad character shift P.

Step 2: Compute a good suffix shift Q.

Step 3: Compare the pattern from right to left by placing the pattern from beginning of text.

Step 4: Repeat steps until pattern goes beyond last character of text or matching.

The Boyer – Moore algorithm is considered as one of the best pattern matching algorithm with its best case time complexity as O $(\frac{n}{m})$. The average and worst case time complexities are given as O(m+n) and O(mn).

Space complexity is given as O(m + k) where, k is the number of different possible characters.

With the Boyer- Moore algorithm analyzed, let us move on to hashing.

Activity 1
Construct a good suffix table for the pattern TOMATO

Self Assessment Questions

4. Input enhancement is to _____ the input pattern.

5. In Horspool's algorithm, the characters from the pattern are matched _____.
6. The two heuristics in Boyre-Moore algorithm are _____ and _____.

9.4 Hashing Methodology

Hashing is the method by which a string of characters or a large amount of data is transformed into a usually shorter fixed-length value or key that represents the original string. This key is used to index and retrieve items from a database. We can find items faster using the shorter hashed key than by using the original value.

Hashing is performed on arbitrary data by a hash function. The code generated is unique to the data it came from.

9.4.1 Hash function

What is hash function?

A hash function is a function that converts data to either a number or an alphanumeric code. The hashing algorithm is called the hash function.

Hash functions are primarily used in hash tables, to locate a data record given its search key. The hash function maps the search key to the hash. The index gives the place where the corresponding record is stored. Therefore, each slot in a hash table is related with a set of records, rather than a single record. Each slot in a hash table is called a *bucket*, and hash values are also called *bucket indices*.

A hash function returns hash values, hash codes, hash sums, checksums or simply hashes. The hash function hints at the record's location – it tells where one should start looking for it.

Uses of Hash functions

Hash functions are used to speed up table lookup or data comparison tasks – such as finding items in a database, detecting duplicate or similar records in a large file, and so on. Hash functions are also used to determine if two objects are equal or similar, checksums over a large amount of data and finding an entry in a database by a key value. The UNIX C-shell uses hash table to store the location of executable programs.

9.4.2 Collision resolution

Hash collisions are unavoidable when hashing a random subset of a large set of possible keys. A hash function can map two or more keys to the same hash value. In many applications, it is desirable to minimize the occurrence of such collisions, which means that the hash function must map the keys to the hash values as evenly as possible. Therefore, hash table implementations have some collision resolution strategy to handle such events. The most common strategies are:

- Open Hashing (Separate Chaining)
- Closed Hashing (Open Addressing)

The keys (or pointers to them) need to be stored in the table, together with the associated values in this method.

Load factor

The performance of collision resolution methods does not depend directly on the number *n* of stored entries, but depends strongly on the table's *load factor*, the ratio *n/s* between *n* and the size *s* of its bucket array.

Separate chaining

Hash collision is resolved by separate chaining also called *open hashing* or *closed addressing.*

In this strategy, each slot of the bucket array is a pointer to a linked list which contains the key-value pairs that are hashed to the same location. Lookup scans the list for an entry with the given key. Insertion involves adding a new entry record to both the ends of the list belonging to the hashed slot. Deletion involves searching the list and removing the element.

Chained hash tables with lists which are linked is popular because they require only basic data structures with simple algorithms, and can use simple hash functions that are unsuitable for other methods.

A chained hash table is effective when the number of entries *n* is much higher than the number of slots. The performance decreases (linearly) with the load factor. For example, a chained hash table with 100 slots and 1000 stored keys (load factor 10) is five to ten times slower than a 1000-slot table (load factor 1); but still 1000 times faster than a plain sequential list, and possibly even faster than a balanced search tree.

In separate-chaining, the worst-case scenario is when all entries are inserted into the same bucket, in which case the hash table is ineffective and the cost is that of searching the bucket data structure. If the bucket data structure is a linear list, the lookup procedure may have to scan all its entries; hence the worst-case cost is proportional to the number *n* of entries in the table.

Open addressing

Hash collision resolved by open addressing is called closed hashing. The term "open addressing" indicates that the location ("address") of the item is not determined by its hash value.

In open addressing, the entry records are stored in the bucket array itself. When a new entry has to be made, the bucket is examined, starting with the hashed-to slot and proceeds in some *probe sequence*, until an unoccupied slot is found. When searching for an entry, the buckets are scanned, until either the target record is found, or an unused array slot is found, which indicates that there is no such key in the table. The popular probe sequences are:

- ***Double hashing*** – the interval between probes is computed by another hash function.
- ***Linear probing*** – the interval between probes is fixed (usually 1).
- ***Quadratic probing*** – the interval between probes is increased by adding the successive outputs of a quadratic polynomial to the starting value given by the original hash computation.

A drawback to the open addressing schemes is that the number of stored entries cannot exceed the number of slots in the bucket array. In fact, even with good hash functions, their performance decreases when the load factor grows beyond 0.7 or so.

Open addressing schemes also put more strict requirements on the hash function. The function must distribute the keys more uniformly over the buckets and minimize clustering of hash values that are consecutive in the probe order.

Open addressing saves memory if the entries are small (less than 4 times the size of a pointer). Open addressing is a waste if the load factor is close to zero (that is, there are far more buckets than stored entries), even if each entry is just two words.

In the next section we will study the indexing schemes.

Activity 2
Compare the collision resolution strategies and list the disadvantages of one strategy over the other

Self Assessment Questions

7. Each slot of a hash table is often called a _____.
8. Collision occurs when a hash function maps two or more keys to the _____.
9. When the interval between probes is computed by another hash function it is _____.

9.5 Indexing Schemes

An effective indexing scheme will help in easy retrieval of data. Different tradeoffs are involved in different indexing techniques. An indexing scheme which is faster may require more storage. The B-tree is one such important index organization.

9.5.1 B-Tree Technique

The B-Tree is an individual indexing scheme at the high speed end of the spectrum. The tradeoff here is that in exchange for fast access times, you pay in terms of code and memory buffer size and in disk space for the B-Tree themselves.

It uses an index to the data records. A file that contains enough information to describe the position and key of each record in the data file is built. The index is organized into a branching tree structure and the tree is kept balanced. The number of index accesses to reach a record is proportional to the logarithm of the number of records in the file. That is to say access time increases slowly as the number of records increases. As the branching factor increases, the height of the tree decreases, thus making access quicker.

A B-Tree of order *m* (the maximum number of children for each node) is a tree which satisfies the following properties:

- Every node has at most m children.
- Every node (except root) has at least m⁄2 children.

- The root has at least two children if it is not a leaf node.
- All leaves appear in the same level, and carry information.
- A non-leaf node with *k* children contains *k*−1 keys.

Internal nodes in a B-Tree – nodes that are not leaf nodes – are represented as an ordered set of elements and child pointers. Every internal node has maximum of *U* children and – other than the root – a minimum of *L*. The number of elements is also restricted for the leaf nodes but these nodes have no children or child pointers.

Root nodes have an upper limit on the number of children but no lower limit. For example, when there are less than *L*−1 elements in the entire tree, the root will be the only node in the tree, and it will have no children at all.

B-Trees assign values to every node in the tree, and the same structure is used for all nodes. However, since leaf nodes never have children, a specialized structure for leaf nodes in B-Trees will improve performance.

Search

Searching is similar to searching a binary search tree. Starting from the root, the tree is recursively traversed from top to bottom. At each level, the search chooses the child pointer (sub tree) whose separation values are on either side of the search value.

For example consider the figure 9.8. To search the number 5 in the tree we start at the root node and traverse through the tree. The number 5 is compared with root node value 13 and since it is less than 13 the searching operation shifts to the left sub-tree. At the second level in the tree it again compares with both the numbers 4, 7. The number 5 is between 4 and, so it moves to the second element 7. The comparison continues in this way and the node having number 5 is found.

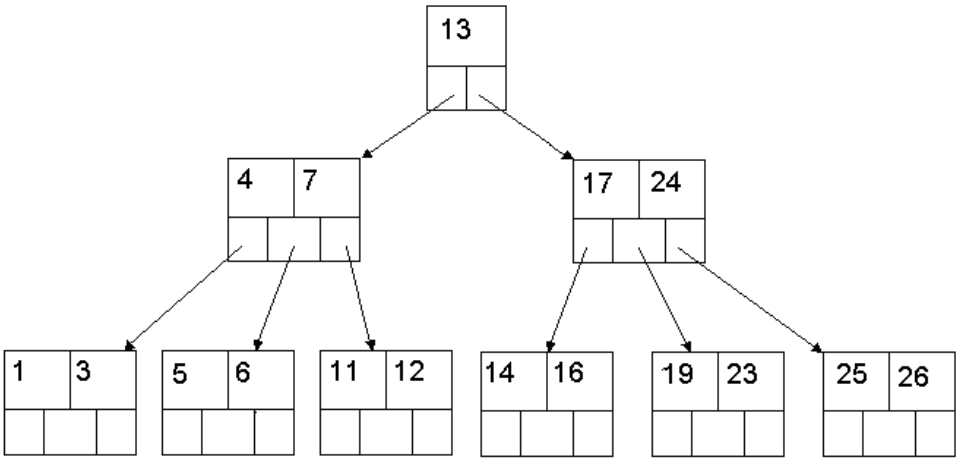


Figure 9.8: B-tree Example for Search Operation

Binary search is typically used within nodes to find the separation values and child tree of interest.

Insertion

To understand the process of insertion let us consider an empty B-tree of order 5 and insert the following numbers in it 3, 14, 7, 1, 8, 5, 11, 17, 13. A tree of order 5 has a maximum of 5 children and 4 keys. All nodes other than the root must have a minimum of 2 keys. The figure 9.9 shows how the 1st four numbers gets inserted.

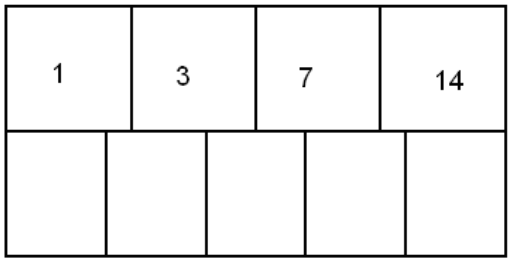


Figure 9.9: Insertion of First Four Numbers in the B-Tree

To insert the next number 8, there is no room in this node, so we split it into 2 nodes, by moving the median item 7 up into a new root node as depicted in figure 9.10.

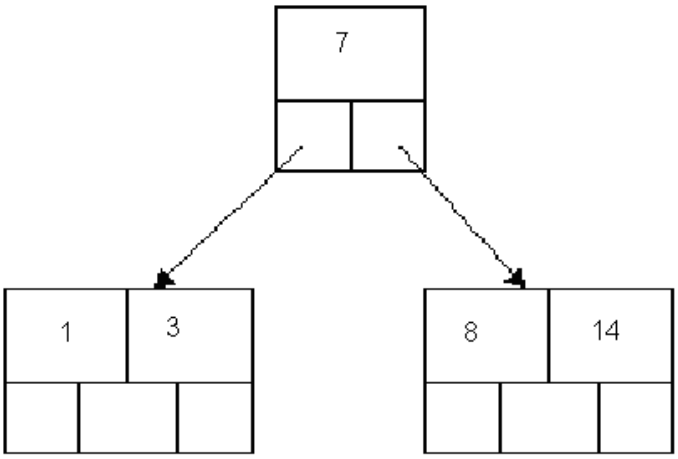


Figure 9.10: B-Tree After Inserting Number 8

Insertion of the next three numbers 5, 11, and 17 proceeds without requiring any splits. This is illustrated in figure 9.11.

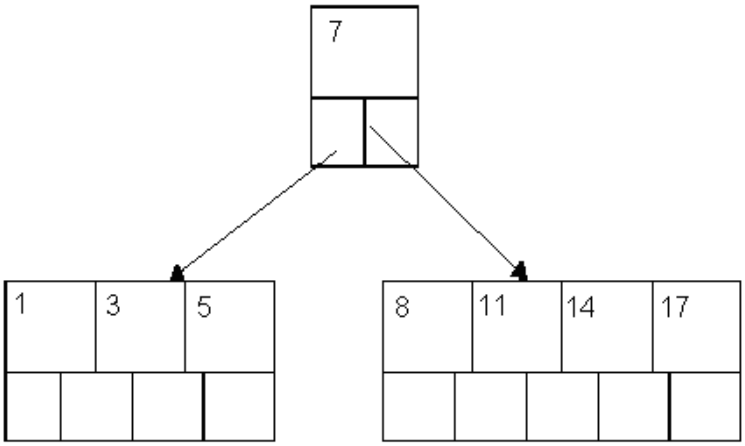


Figure 9.11: B-Tree After Inserting Numbers 5, 11 & 17

A split is to be done to insert 13. 13 is the median key and so it is moved up into the parent node. This is depicted in figure 9.12.

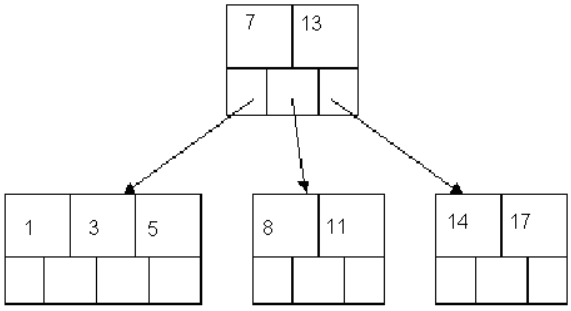


Figure 9.12: B-Tree After Inserting Number 13

Deletion

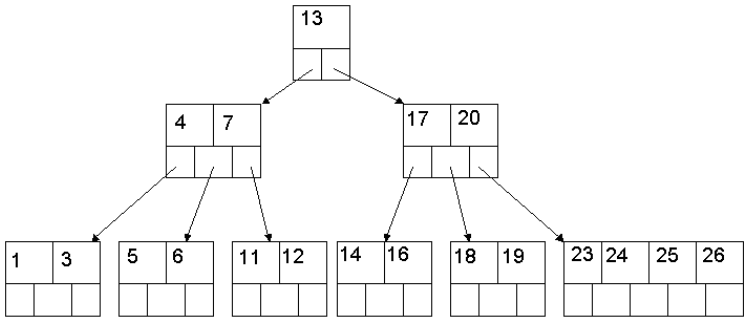


Figure 9.13: B-Tree Before Deletion

Considering the B-tree in figure 9.13, let us understand the process of deletion by the deleting the following numbers one by one

To delete 20, we find its successor 23 (the next item in ascending order) because it is not a leaf node, and move 23 up to replace 20. Therefore we can remove 23 from the leaf since this leaf has extra keys.

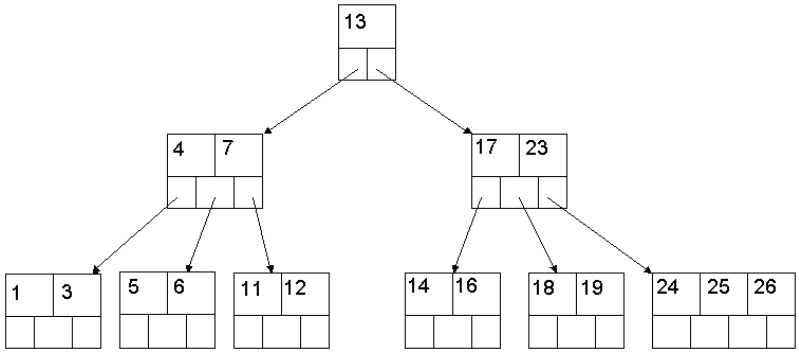


Figure 9.14: B-Tree After Deleting Number 20

Next, to delete 18, even though 18 is in a leaf, we can see from figure 9.6 that this leaf does not have an extra key; the deletion results in a node with one key, which is not acceptable for a B-tree of order 5. If the sibling node to the immediate left or right has an extra key, we then borrow a key from the parent and move a key up from this sibling. In this specific case, the sibling to the right has an extra key. So, the successor 23 of 19, is moved down from the parent, and the 24 is moved up, and 19 is moved to the left so that 23 can be inserted in its place.

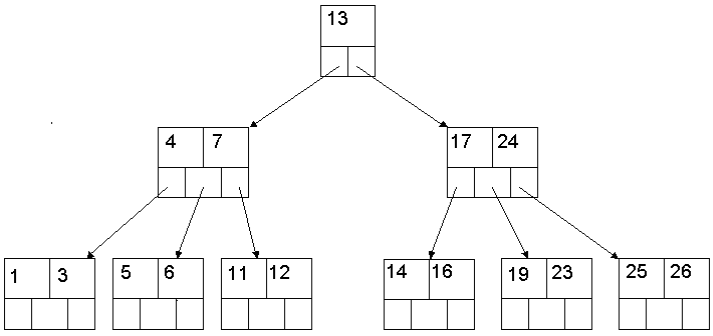


Figure 9.15: B-Tree After Deleting Number 18

To delete 5, there is another problem now. Refer to figure 9.15. Although 5 is a leaf, the leaf has no extra keys, nor do the siblings to the immediate right or left. In this case the leaf has to be combined with one of the two siblings. This results in moving down the parent's key that was between those of these two leaves. Therefore let's combine the leaf containing 6 with the leaf containing 1 3 and move down the 4. This results in an invalid B-tree as shown in figure 9.16.

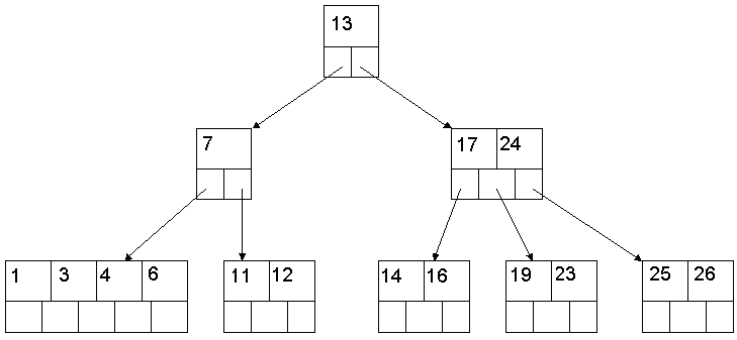


Figure 9.16: Invalid B Tree After Deleting 5

Now the parent node contains only one key, 7 which is not acceptable. If this node had a sibling to its immediate left or right having a spare key, then we could again "borrow" a key. Suppose the right sibling (the node with 17 24) had one more key. We would then move 13 down to the node with too few keys and move the 17 up where the 13 had been. The 14 and 16 nodes would be attached via the pointer field to the right of 13's new location. Since in our example we have no way to borrow a key from a sibling, we must again combine with the sibling, and move down the 13 from the parent. In this case, the tree's height reduces by one. The resulting B-tree is shown in figure 9.17.

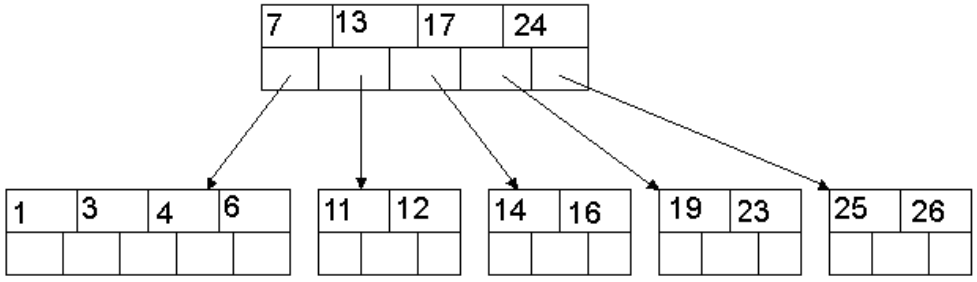


Figure 9.17: B-Tree After Deleting 5

Self Assessment Questions

10. As the _____ increases the height of the tree decreases thus speeding access.
11. Access time increases slowly as the number of records _____.
12. The insertions in a B-Tree start from a _____.

9.6 Summary

Let us summarize the unit here.

We usually analyze the efficiency of an algorithm in terms of its time and space requirements. Usually the efficiency of an algorithm is stated as a function relating the input length to the number of steps (time complexity) or storage locations (space complexity).

Distribution counting is an input enhancement method wherein a separate array is used to store the information generated during the sorting process and these arrays enhance the sorting process. Horspool's and Boyre-Moore algorithms are string matching algorithms where in the pattern is compared

with the text and shifting of the pattern is done by computing shift size. Thus searching operation becomes faster.

Hashing is a technique that uses a hash key to find items. Collision occurs when the hash key value of two items turns out to be the same value. This is rectified by the two collision resolution methods - separate chaining and open addressing. The branching factor in B-Tree technique speeds the access time.

Thus by using all the above techniques, we can reduce the execution time of an algorithm.

9.7 Glossary

Term	Description
Leaf node	A leaf node is a node in a tree data structure that has no child nodes.
Linked list	A linked list is a data structure which consists of a sequence of data records, that in each record there is a field that contains a reference

9.8 Terminal Questions

1. Explain distribution counting with an example.
2. Explain the two types of collision resolution in hashing.
3. Describe the algorithms based on input enhancement in string matching.
4. What is a hash function?
5. How does B-Tree technique enhance space and time tradeoff?

9.9 Answers

Self Assessment Questions

1. Preprocessing
2. Distribution
3. Time efficiency
4. Preprocess
5. Right to left
6. Good suffix and bad character shift
7. Bucket
8. Same hash value

9. Double hashing
10. Branching factor
11. Increases
12. Leaf node

Terminal Questions

1. Refer to 9.2.1 – Distribution counting
2. Refer to 9.4.2 – Collision resolution
3. Refer to 9.3 – Input enhancement in string matching
4. Refer to 9.4.1 – Hash function
5. Refer to 9.5.1 – B-Tree technique

Reference

- Puntambekar, A.A. (2008). *Design and Analysis of Algorithms*, First edition, Technical publications, Pune.
- Donald Adjeroh., & Timothy Bell., & Amar Mukherjee (2008). *The Burrows-Wheeler transform*, Springer Publishing Company.

E-Reference

- www.cs.ucr.edu/~jiang/cs141/ch07n.ppt
- http://documentbook.com/horspool-ppt.html
- http://www-igm.univ-mlv.fr/~lecroq/string/node14.html