Exploring Search Options

Searching is a widely used process in computer applications, primarily to determine whether an element with a particular value is present in a vector or list of elements or not. It acts as a substitute in case of deletions, as without searching an element of a particular value, deletion operations cannot take place. A search can be an evaluation of finding an element (exact match) in a set of given elements, or finding a group of elements (range match) which falls under a certain range of values. In a search operation, the location of the element is also determined. The location can be used later in deletion operations. A search is said to be successful if the element of a particular key value is found in the given vector (or list), and is said to be unsuccessful if the element of a particular key value is not found in the given vector (or list). This chapter shall cover concepts of sequential search operations and direct access by key value (hashing) search operations.

This chapter deals with search operations carried on both vectors and lists (including linked lists). The topics covered are as follows: Searching unsorted and sorted vectors Self-organizing lists Hashing

The approaches discussed in the first two sections are more effective in implementation while using an in-memory-(single node) based system, and the third approach discussed is more effective in implementation while using either in-memory-(single node) or disks (multiple nodes) based systems.

Page 151

﻿

Exploring Search Options

Searching unsorted and sorted vectors

Vectors are simple and widely used data structures used to perform search operations in R. The simplest form of search operation performed on vectors is a linear search or sequential search. In a linear search, each element is compared sequentially within the vector, and then suitable insertions or deletions are performed. Consider an element, S which is to be searched in an unsorted vector, V of length n (indexed from 1 to n ). If the element S is not present in vector V , then a minimum of n comparisons are performed, and if it is present in V at position i , then a minimum of i comparisons are performed. In both scenarios, the number of comparisons is linear, resulting in O(n) as the functional form of system runtime for sequential search in the worst case scenario. The following R code performs a linear search of element S in a vector V of length n :

Sequential\_search <- function(V,S,n) {

i=1

present = FALSE

while(i <= n & present==FALSE) {

if(V[i] == S)

present=TRUE else i = i+1

}

return(list(present=present,key=i))

}

Figure 6.1 illustrates two different sequential operations. Part (a) represents the situation wherein the element S is not present, and Part (b) represents the presence of element S . Each step corresponds to a comparison, regardless of whether the current element in the vector is equal to S or not:

Figure 6.1: Sequential or linear search

[ 140 ]

Page 152

﻿

Chapter 6

Sequential search becomes a bottleneck when n is a very large value. One way to deal with large vectors is to pre-process them using multiple sorting techniques learned previously, and then perform a sequential search. If the current element in vector V is larger than the element S , then the search operation can be terminated early. This reduces the system runtime; however, the asymptote of the search operation does not improve beyond O(n) for the worst-case scenario. The following R code performs a sequential search on an ordered vector V of length n :

Seq\_ord\_search <- function(V,S,n) {

i=1

present = FALSE

while(i <= n & present==FALSE ) {

if(V[i] == S)

present=TRUE else if(V[i] > S)

stop("element S not found") else i=i+1

}

return(present)

}

Figure 6.2 illustrates a sequential search operation being performed on an ordered vector in which the element S is not present. Here, each step corresponds to two comparisons: One, whether the current element is equal to S or not, and the other, whether the current element is greater than S or not (provided S is not equal):

Figure 6.2: Sequential or linear search performed on an ordered vector

Now, consider a situation in which the element S is directly compared with the element in the third position and is found to be greater; then, it becomes imperative that element S is greater than the elements in the first and second position, without even comparing explicitly. This kind of comparison performed on intermittent elements (best possible jumps) rather than on each element of the vector is a key feature of the jump search algorithm.

[ 141 ]

Page 153

﻿

Exploring Search Options

A jump search algorithm is an improvisation on the current sequential search algorithm performed on sorted vectors. Here, the element S is initially compared with elements of the vector V positioned at regular intervals, i . In other words, the element S is initially compared with V[i] , V[2i] , V[3i] , and so on, till the condition of S being lower is met. Once the position wherein the element in that position (key value) is greater than the element S is determined, then the sequential search is performed on its previous i-1 elements. That is, if the element S is lesser than the element V[3i] , then the sequential search is performed on elements between V[2i] and V[3i] . This dividing of the vector into sub-vectors and then performing search operations is similar to the concept of divide and conquer discussed in the Merge sort section in Chapter 5 , Sorting Algorithms . The best possible i for a vector of length n is . The following R code performs a jump search algorithm on a sorted vector V of length n . The jumps are performed at an interval of :

Jump\_search <- function(V,S,n) {

jump <- floor(sqrt(n)) present = FALSE i=1

while(jump < n & V[jump] < S) {

i=jump

jump = jump+floor(sqrt(n)) if(jump>=n)

stop("element S not found")

}

while(V[i] < S & i <= jump) i = i+1

if(V[i]==S) present=TRUE

return(present)

}

[ 142 ]

Page 154

﻿

Chapter 6

Figure 6.3 illustrates the working of a jump search algorithm. Initially, jumps are performed till the current value of the jumped position is greater than the element S , and then linear search is performed within the sub-vector (here, elements in positions 4 , 5 , and 6 ):

Figure 6.3: Jump search algorithm

The jump search algorithm can also be modified such that the jumps are performed at two stages, and sequential search is performed at the third stage. In a sense, a sub-vector is first determined based on a certain jump; then, within that sub-vector, smaller jumps are performed to determine a sub-sub-vector within which the sequential search is performed. This can be generalized using recursive implementation of generating sub-vectors till a single element in the vector V is left out for comparison. This generalized jump search implementation is equivalent to a binary search algorithm. The approach of binary search is to jump directly toward the element in the middle of the vector, and then compare S with it. If the value of S is greater than the middle element, then jump backward, otherwise, jump forward. The jump is always performed toward the middle element of any sub-portion of the vector in consideration. The asymptote of a binary search algorithm for an average case scenario is O(log n) .

A binary search algorithm can be implemented recursively or iteratively. However, recursive implementation can sometimes be risky. Following is the R code implementation of the binary search algorithm:

Recursive implementation (returns the position of the element S if found in V):

Bin\_search\_recursive <- function(V,S,l,h) {

if ( h < l ) {

stop("h should be more than l")

} else {

m <- floor((l + h) / 2) if ( V[m] > S )

Bin\_search\_recursive(V, S,l,m-1) else if ( V[m] < S )

Bin\_search\_recursive(V, S, m+1, h) else

return(m)

} }

[ 143 ]

Page 155

﻿

Exploring Search Options

Iterative implementation (returns whether element S is present in the vector V or not):

Bin\_search\_iterative <- function(V, S,n) {

l=1 h=n

i = 0

while ( l <= h ) {

m <- floor((l + h)/2) if ( V[m] > S ) h <- m - 1

else if ( V[m] < S ) l <- m + 1

else if(V[m]==S) return(TRUE)

}

return(FALSE)

}

Figure 6.4 illustrates the implementation of binary search on a sorted vector V :

Figure 6.4: Binary search algorithm

In any of the algorithms covered so far, the distribution of key values (elements of the vector) has not been considered for search operations. Suppose you need to search a word algorithm in a dictionary. The first step will be to look for all the words which begin with the letter a , and then, within those, search for words which begin with al , and so on, till the word algorithm is found. In other words, the distribution of the word is taken into account before computing the next steps of where to search in the dictionary. This form of search, in which the knowledge of element S is considered before computing the next search steps, is called dictionary search or interpolation search. The first search position ( p ) for the element S in the sorted vector V is computed as follows:

[ 144 ]

Page 156

﻿

Chapter 6

Upon computing the primary search position p , the element S is compared with V[p] . If S equals V[p] , then the search operation is terminated; otherwise, the position p is used to split the vector into two sub-vectors. Based on the value of S with respect to V[p] , the search continues in either of the sub-vectors, as illustrated in Figure 6.5 . The split at position p is similar to the split in the binary search algorithm. Again, the new position of the sub-vector is computed based on the distribution of elements within it, and the search operation continues till the element S is found or the vector is narrowed until no elements are left. The system runtime reduces considerably, which follows an asymptote of O(log log n) for average case scenarios.

The following R code performs an interpolation search of element S in the sorted vector V of length n :

Interpolation\_search <- function(V,S,n) {

i=1; j=n; l=V[1]; h=V[j]; if(S<l | S>h) return(FALSE) while(i < j) {

k = floor(i+((j-i)\*(S-l))/(h-l)) split = V[k] if(S>split){

i=k+1; l=split

}else if(S < split){

j=k-1; h=split

}else if(V[k]==S){

return(TRUE)}

}

return(FALSE)

}

Figure 6.5 illustrates the working of an interpolation search:

Figure 6.5: Interpolation or dictionary search algorithm

[ 145 ]

Page 157

﻿

Exploring Search Options

Self-organizing lists

So far, we have learned that the performance of search can be enhanced by sorting the vectors based on their key values prior to the search operations. However, there seems to be an another approach for sorting vectors, which is not based on key values but on the expected frequency of accessing the key values for comparison purposes. This kind of sorting based on expected frequency of access can sometimes be cheaper compared to sorting based on key values, thereby increasing the performance of search operations. Consider a vector V sorted based on the frequency of access of key values, but not on the value of its elements. In other words, the elements with a higher probability ( p i ) of getting compared with the search element S is placed first, followed by the element with the second highest probability, and so on. The search for element S is performed sequentially on all the sorted elements in the vector. Upon multiple search iterations, the expected number of comparisons required for one search is given as follows:

Here, the cost to access the first element is 1 (only a single element to compare with S ), and the probability of access of this first element is p 1 . Similarly, the cost to access the second element is 2 (as the first and second elements need to be compared with, and its probability of access is p 2 , and so on. Assuming the possibility of search on each of the elements in the vector, the sum of all the probabilities from p 1 to p n is infinity.

This approach of sorting based on the frequency of comparisons has some disadvantages. Primarily, it is very difficult to determine the probabilities of access in advance if the corresponding vector has not been iterated over a bunch of search elements. Moreover, the records which had a higher frequency of access initially may not continue to stay for long time. Thus, the probability of access for some elements might change over time. These constraints led to the concept of self-organizing lists, wherein the pattern of element access is also taken into account along with their frequency of access. These self-organizing vectors are based on heuristics, examples of some heuristics for self-organizing vectors are as follows:

[ 146 ]

Page 158

﻿

Chapter 6

Count : This is the most basic heuristic of self-organizing vectors. Here, the count refers to the number of comparisons or accesses being made with the elements of the vectors. The count of each key value is stored, which is used in ordering of the vector. In parallel, the elements are also moved toward the left of the vector as its count starts increasing more than the elements preceding it. The main drawback of this heuristic is that the ordering is very hostile to the change in frequency of access over time. In other words, once the element gets a higher count, it nearly always remains toward the left of the vector regardless of further changes to other elements' counts. Also, this method requires additional memory to store the count information.

Move-to-front : In this heuristic, once the element S is found, the corresponding element in the vector is moved toward the first position, and all the other elements are pushed back by one positon. This kind of heuristic is called move- to-front. This is easy to implement in linked lists against vectors. In vectors, bringing a near-end element toward the front of the vector requires the

displacement of a large number of preceding elements. The cost of move-to-front is almost twice the cost required by the count heuristic, wherein n searches are performed on the vector and the elements are, accordingly, order based on the frequency of access. It performs better in scenarios where the elements are accessed frequently for a brief period of time, as these elements will be near the front of the vector during this period of access. However, it performs poorly when elements are processed repeatedly in a sequential order.

Transpose : The heuristic of swapping adjacent elements based on their frequency of access is termed as transpose. It performs well in both linked lists and vectors. The transpose will inherently move the most frequently accessed elements to the front of the vector. The elements which were initially accessed frequently and moved to the front will start to slowly drift backward once they are no longer accessed frequently. Thus, it performs well for scenarios where there is change in the frequency of access. In some situations, it performs poorly. Assume a sequence of search operations wherein the last and second-last elements are accessed alternately. Then these elements will get swapped for each iteration, but neither of them will move toward the front of the vector. However, these kinds of situations are rare. This can be resolved if the accessed elements are moved forward by some fixed number of positions instead of swapping with their adjacent preceding element.

[ 147 ]

Page 159

﻿

Exploring Search Options

Now let's understand each heuristic using an example. Consider a numeric vector, V of eight elements arranged in an order of key values, as shown in the following diagram:

Figure 6.6: Example of numeric vector arranged in order of key values

Now, let's perform a series of 12 search operations in the following order of elements ( S ):

Figure 6.7: Vector of search operations to be performed

Heuristic 1 – Count

In count based heuristic, the frequency of access of the elements start moving forward. After the first three searches, the element 6 will be first, followed by 4 , and so on. The total cost of all these accesses will be 45 comparisons. The following R code implements self-organizing lists, and returns a sorted vector based on sequential search. The input is a vector V , the sequence of search elements is S , the number of elements to be searched (that is, length of S ) is n\_search , and the number of elements in the input vector (that is, the length of V ) is n :

SOL\_count <- function(V,S,n\_search,n) {

if(is.null(V)) stop("NO elements in input vector") if(is.null(S)) stop("NO elemens to search") i=1

count <- as.list(sapply(1:n,function(x) 0)) names(count) <- V cl <- class(V)

while(i<=n\_search) {

if(Sequential\_search(V,S[i],n)$present){ key <- Sequential\_search(V,S[i],n)$key count[key][[1]] <- count[key][[1]] + 1 count <- count[order(-unlist(count))] V <- as.numeric(names(count))

}

i=i+1

}

return(V)

}

[ 148 ]

Page 160

﻿

Chapter 6

The final sorted vector based on count is as follows:

Figure 6.8: Output from Heuristic 1 – Count

Heuristic 2 – Move-to-front

Here, upon finding the element S , the element is moved toward the front of the vector. The following R code implements the move-to-search heuristic of self-organizing lists, and returns a sorted vector. The input is a vector V , the sequence of search elements is S , the number of elements to be searched (that is-length of S : n\_search ) and number of elements in the input vector (that is, the length of V : n ):

SOL\_move <- function(V,S,n\_search,n) {

if(is.null(V)) stop("NO elements in input vector") if(is.null(S)) stop("NO elemens to search") i=1

while(i<=n\_search) {

if(Sequential\_search(V,S[i],n)$present){ if(Sequential\_search(V,S[i],n)$key !=1){ key <- Sequential\_search(V,S[i],n)$key temp <- V[key] V <- V[-key]

V <- c(temp,V)

} }

i <- i+1

}

return(V)

}

[ 149 ]

Page 161

﻿

Exploring Search Options

The total cost of these accesses will be 54 comparisons, and the final sorted vector is as follows:

Figure 6.9: Output from Heuristic 2 – Move-to-front

Heuristic 3 – Transpose

Here, the elements, once found, are transposed with the adjacent element till it moves toward the front. The following R code implements the transpose heuristic of self- organizing lists, and returns the sorted vector. The input is a vector V , the sequence of search elements is S , the number of elements to be searched (that is, the length S ) is n\_search , and the number of elements in the input vector (that is, the length of V ) is n :

SOL\_transpose <- function(V,S,n\_search,n) {

if(is.null(V)) stop("NO elements in input vector") if(is.null(S)) stop("NO elemens to search") i=1

while(i<=n\_search) {

if(Sequential\_search(V,S[i],n)$present){ if(Sequential\_search(V,S[i],n)$key !=1){ key <- Sequential\_search(V,S[i],n)$key temp <- V[key-1]

V[key-1] <- V[key] V[key] <- temp

} }

i <- i+1

}

return(V)

}

[ 150 ]

Page 162

﻿

Chapter 6

The total cost of these accesses will be 62 comparisons, and the final sorted vector is as follows:

Figure 6.10: Output from Heuristic 3 – Transpose

The asymptote of self-organizing lists based on system runtime is O(log n) , which is similar to binary search trees; however, the former performs better in many scenarios. The main advantage of self-organizing lists is the non-requirement of a pre-sorted vector, as sorting itself requires a certain cost. Also, the cost to insert a new element is also low, as its position need not be determined, which is compulsory in the case of insertion in a sorted vector. Self-organizing lists are simple to implement, and show better performance even for smaller vectors (or lists). Thus, with a minor change in the algorithm, the performance of a sequential search can be enhanced using self-organizing lists without any prerequisite for a sorted vector.

Hashing

Hashing is a different type of search technique, wherein the key values of the vector are directly accessed. It is a process of searching the element using some computation to map its key value to a position in the vector. The values of the vector can be placed in any order satisfying certain calculations rather than being based on any key value or their frequency of access. The function which performs the computation to map the key values to positions in a vector is termed a hash function, also denoted by h . The vector which holds the elements after the necessary calculations is termed a hash table (also denoted by HT ), and its positions (or keys) are termed slots. The number of slots in a hash table is denoted by m , and each slot is numbered between 1 and m . The key objective of hashing is to arrange the elements in a hash table HT such that for any key value K and some hash function h , the slot in the hash table is defined as i = h(K) , where i lies between 1 to m and the element in HT[i] is K . The concept of hashing can be explained using ISBN book codes. In a library, each book is assigned a unique ISBN code, which is used for searching for books. The books form the raw elements (key values) of the vector, and the library is its hash table. The hash function is used to allocate the ISBN code for each book, and the codes serve as slots of the hash table.

[ 151 ]

Page 163

﻿

Exploring Search Options

The key purpose of hashing is to determine whether an element belongs to a particular vector or not. It is not suitable for situations where multiple elements have the same key value. It does not support searching elements falling in a certain range, or searching for an element with a maximum or minimum key value. It does not even allow access to the elements in the order of the keys. Hashing supports both in-memory and disk-based searching, and it is one of the widely used approaches for organizing large databases stored on disks.

Let's begin with a simple scenario, with each record being assigned to each unique key value upon hashing. Although it is very rare, here, the hash table HT can be generated directly using the key values as its slots. In other words, the element k is assigned to HT[k] using a simple hash function h(k) = k .

Now, let's consider scenarios that occur more often. The scenario in which two elements k 1 and k 2 are assigned to the same slot α using the hash function h , that is – h(k 1 ) = h(k 2 ) = α , is termed collision. This is generally avoided by using different forms of hash functions and increasing the slots in the hash table. Suppose the number of slots is increased much higher compared to the number of elements that need to assigned. In this case, there is a risk of skewed distribution. Also, the hash table will be left with many empty slots, thus requiring a lower number of slots in the hash table sufficient to handle all the elements without any skewed distribution.

Following is a two-step procedure devised to find the element K in a hash table HT generated using a hash function h :

1. Compute the slot in the hash table using the hash function h(K)

2. Search for the element with value K starting from the slot h(K) such that the

likelihood of collision is minimized

Hash functions

Technically, any function which is used for distributing the elements (key values) of a vector (or a list) into a hash table is termed a hash function. These functions perform direct mathematical operations on the elements, and the corresponding output is then used for assigning the elements to the respective slots in the hash table. Quite often, the elements belong to a much larger numeric range compared to the range of slots in the table. Due to this kind of reduction, the possibility of the elements being assigned to a single slot increases, leading toward collision. Consider a group of 50 employees in an organization. Now, what is the probability that two employees will have the same birthday?

[ 152 ]

Page 164

﻿

Chapter 6

One of the key objectives of hash functions is to minimize the number of collisions. Hashing with zero collisions is termed perfect hashing. In perfect hashing, no two elements are assigned to a single slot of the hash table. This can be made possible if we have all the elements of the input vector prior to selecting the hash function. Once the hash table is generated using this hash function, the elements can be accessed directly without any further search required within each slot. Selecting this kind of perfect hash function can be very expensive, but it can be a trade-off toward achieving extremely efficient search performances.

Along with minimizing collisions, hash functions should also account for minimizing the number of slots in the hash table. It is advisable not to have many empty slots in the hash table, as it occupies unnecessary memory. However, it is highly unlikely to achieve zero collisions with all elements (key values) assigned to all the slots of the generated hash table. In a nutshell, a hash function can assign all elements to a single slot of the hash table, or generate a unique number of slots for each corresponding element. It is always desirable to select the hash function in such a way that each slot of the hash table has equal probability of getting filled with any key value of the input vector. But it is implausible to have control over the key values of the input vector. The efficiency of the hash function depends on how it distributes the key values within the allowable range of slots (or keys) of the hash table. If the key values of the input vector are a random selection of numbers uniformly distributed within a key range, then any hash function that generates slots within this key range with equal probability of elements being assigned to it will also uniformly distribute the input key values in the hash table. In such scenarios, the input key values are well distributed across the hash table. However, in most scenarios, the input key values are highly skewed toward a smaller range or poorly distributed across the key range. This makes it more difficult to devise the hash function which can uniformly distribute these skewed key values into a hash table. This can sometimes be minimized if the distribution of input key values is known in advance.

The following are some reasons for non-uniformity observed among the input key values:

If the input key values are a natural frequency of occurrences, then they are highly likely to follow a Poisson distribution. In other words, only a few key values occur more often, and many others occur relatively rarely. For example, consider the number of Internet connections across the country. The number of connections in urban areas is quite high compared to the number of connections in rural areas. Also, the number of rural areas is quite high compared to the number of urban areas. Therefore, the distribution of Internet connections is highly skewed toward a lesser number of areas (urban) across the country. Sometimes, data collection can be skewed due to improper adoption of sampling techniques.

[ 153 ]

Page 165

﻿

Exploring Search Options

Thus, the distribution of input key values plays a vital role in designing the hash functions. If the distribution of input key values is unknown, then select the hash function

appropriately such that the key values are distributed across the hash table, avoiding any undue skewedness. If the distribution of input key values is known, then select the hash functions based on its distribution, thereby avoiding any undue skewedness. The following hash function performs hashing of integers into a hash table of size 18 :

hash\_int <- function(K) {

return (K %% 18)

}

The following hash function performs hashing on strings of characters using the folding approach:

hash\_string <- function(K,n,M) {

hashValue <- 0 for(i in 1:n){

hashValue <- hashValue+as.numeric(charToRaw(substr(K,i,i)))

}

return(hashValue %% M)

}

In this preceding function, the ASCII values of each character in a string are added, which is then used to derive a slot key of hash table. Thereby, the order of characters in a string plays no role in deciding the value of slot keys. Generally, a hash table of smaller size tends to perform better as all the characters within a string are given equal weights (irrespective of their order), and this, in turn, helps ensure a uniform distribution of strings across the slots in the hash table. Similarly, this can also be adopted for the hashing of integers (by adding the digits of integers). However, an underlying assumption is that there are no integers that can skew the results considerably (such as 14, 41, 50, 5, 23, 32; and they each will be assigned to the slot key of value 5), and generate hash keys much larger than the size of the hash table ( M ). As a final step, the modulus operation is performed on the summed up values to obtain values of the slot keys in the range of 0 to M-1 . A good distribution of slots primarily depends on the range of expected summations of each string. For example, the summation range of strings of length 5 (all uppercase) will be in the range of 325 to 450, as the ASCII value of A is 65 and Z is 90. As the range is not very spread out, larger hash tables tend to show more skewed distributions, and the key values are not evenly distributed across slots.

[ 154 ]

Page 166

﻿

Chapter 6

In most practical scenarios, collision remains to haunt the implementation of hashing. These collisions can be minimized using certain resolution techniques, such as the following: Open hashing or separate chaining : The collisions are stored outside the hash table

Closed hashing or open addressing : The collisions are stored within the hash table such that one of the colliding key values is stored in another slot of the hash table

Open hashing

In open hashing, slots in the hash table are defined as heads of a linked list, and colliding values are assigned to each slot of the hash table. Figure 6.11 illustrates the working of open hashing. Consider a vector of values 484, 253, 697, 467, 865, 823, 963, and 651, which are hashed into a table with keys 0 to 9 using the hash function: h(K) = K mod 10 . The numbers are inserted into the hash table in the aforementioned order. Collisions are observed at two slots: key 3 and key 7, and the key values within these slots are linked with each other using pointers. Other slots have only single key values.

Once the slot is identified using the hash function, the search operation begins within that corresponding slot. The key values within each slot can be ordered using multiple techniques such as insertion order, key-value order, frequency-of-access order, move-to- front order, or transpose order. In the case of key-value order, the search operation can be efficient, as it culminates once it encounters a key value greater than the search element. Whereas, if the elements within the slot are unordered or ordered using self-organizing techniques, then all the elements within each slot need to be accessed before culminating the search operation for the worst-case scenario (that is, the search element is not present in the slot):

[ 155 ]

Page 167

﻿

Exploring Search Options

Figure 6.11: Open hashing

Suppose the number of input key values is higher than the number of available slots in the hash table, then an ideal hash function is one which distributes all the input key values uniformly among all the available slots. On the other hand, if the number of input key values is less than the number of available slots in the hash table, then an ideal function will distribute in such a way that only one key value is assigned to each slot, avoiding any collision. In the former scenario, the search operation continues post evaluating the slot in the hash table, whereas, in the latter, the search operation culminates once the slot is identified. Thus, the average cost of system runtime for the latter scenario is θ(1) , which is lower than the average cost of the former.

The hash tables can be generated both in-memory (single-node cluster) or on disks (multiple-node cluster), where different slots of the hash table (or linked lists) can be assigned to different node clusters. In the single-node cluster format, all the elements are accessed seamlessly within the same node, whereas in the multiple-node cluster format, different disks need to be accessed before completing the search operation. Open hashing is more suitable for in-memory-based hash tables than disk-based hash tables, as a multi-node cluster defeats the very purpose of hashing, which is to provide seamless access to search particular key values.

One can observe similarities between open hashing and the binsort algorithm. Some of them are listed next:

In a binsort algorithm, the elements of the input vector are initially assigned to multiple bins, and each bin can have multiple elements. Similarly, in open hashing, the input elements are initially assigned to multiple slots of the hash table, and each slot can have multiple elements.

[ 156 ]

Page 168

﻿

Chapter 6

In binsort, the number of elements in each bin is a smaller number and sorting is performed individually on each bin. Similarly, in open hashing, the number of elements assigned to each slot is a smaller number; thereby, fewer accesses are required to complete the search operation.

Closed hashing

In closed hashing, all the input key values are stored within the hash table itself. If any collision arises, a collision resolution policy is adopted. Initially, hashing is performed on each element based on its key value, and its corresponding home slots are identified. While assigning each element to its corresponding home slot, if a new element collides with an already assigned element for a given home slot, then the new element is assigned to another empty surrogate slot based on a collision resolution policy.

This resolution policy is also adopted during search operations, because not all elements are assigned to their respective home slots, and other elements which are assigned to empty surrogate slots can also be recovered to complete the search.

Bucket hashing

Bucket hashing is one of the variants of closed hashing. In bucket hashing, the slots of the hash table are initially grouped into a relatively smaller number of buckets. Suppose there are M slots in a hash table, and there are B buckets, then M/B number of slots are assigned to each bucket. The hash functions are now directly linked to the bucket keys. Initially, the hash function starts assigning key values to the first empty slot in the bucket. In case of a collision, the slots in the bucket are sequentially searched till an empty slot is found. In the worst case, where the bucket gets filled, the elements are assigned to empty slots in an overflow bucket of infinite capacity. The overflow bucket is shared by all the buckets. A good implementation is one where most of the key values are filled in the respective buckets, and very few (kind of outliers) are assigned to the overflow bucket. Figure. 6.12 illustrates the implementation of bucket hashing. Consider a vector of values 484, 253, 697, 467, 865, 823, 963, and 651, which are hashed into a table with buckets 0 to 4 using the hash function h(K) = K mod 5 . Each bucket has two slots along with an overflow bucket, in case the existing bucket gets filled. Upon sequential hashing of the given vector, zero elements are assigned to bucket 1, one element is assigned to bucket 5, two elements are assigned to buckets 1 and 2, and three elements are assigned to bucket 3.

[ 157 ]

Page 169

﻿

Exploring Search Options

As bucket 3 has only two slots, the third element is assigned to the common overflow bucket:

Figure 6.12: Bucket hashing

In case of a search operation, the first step is to determine which bucket the search element can be attributed to, using the hash function h . Then, the elements within this bucket are searched. If the search element is not found inside the bucket and the bucket is not full, then the search operation culminates. In case the bucket is full, the overflow bucket is then searched until the search element is found or all the elements within the overflow bucket have been searched. The search operation can sometimes be time consuming if the overflow bucket is too large.

So far, the key values are hashed to a given bucket with some number of slots. Consider a scenario wherein the key values are hashed to a slot which, in turn, belongs to a bucket. In short, the buckets are indirectly related to the hash function, and the slots in each bucket play a pivotal role. Here, the key values are initially assigned to their respective home slots, which belong to a certain bucket. In case the home slot gets filled up, the slots in the respective bucket are scanned sequentially, and then filled accordingly. Consider a bucket of six slots marked from 0 to 5, with the third slot already filled. Suppose a new element is again assigned to the third slot of the given bucket, then the collision resolution process will begin. In this process, initially the fourth and fifth slots are scanned for any vacancy, followed by the first and second slots. In case all the slots are full, the new element is assigned to an empty slot in the overflow bucket (which has the capacity to hold infinite slots). This approach is advantageous over the former approach, as here, any slot in a bucket can act as a home slot, whereas in the former approach, only the first element of the bucket can act as a home slot. Thereby, the number of collisions is also reduced.

[ 158 ]

Page 170

﻿

Chapter 6

Figure 6.13 illustrates the working of modified bucket hashing:

Figure 6.13: Modiﬁed bucket hashing

Unlike open hashing, bucket hashing is good to implement on multiple disks or nodes. The size of the buckets can be used to determine the size of each node cluster. Whenever a new search or insertion happens, the corresponding bucket is called into memory and all the search/insertion operations occur seamlessly as there is only one node to access. In case the bucket is full, the overflow bucket is pulled into the given node. It is highly recommended to keep the overflow bucket small enough to prevent any unnecessary node accesses.

Linear probing

Linear probing is one of the widely used closed hashing techniques which is devoid of bucketing, and has the potential to access any slot of the hash table using the updated collision resolution policy.

The primary objective of the collision resolution policy is to obtain the free slot in the hash table when any collision occurs, that is, when the home slot of any key value is already filled. The collision resolution approach can be updated such that it generates a sequence of slots which can be orderly filled upon collisions. The first slot of the sequence acts as a home slot, and subsequent slots act as surrogates. In case of a collision, the slots are sequentially scanned till an empty slot is obtained. This sequence of slots is termed a probe sequence, which is generated using a probe function represented p .

[ 159 ]

Page 171

﻿

Exploring Search Options

Similarly, also during the search operation, the same probe function is used to retrieve all the relevant key values which were earlier inserted for a given home slot. One of the key assumptions of the probe sequence generated using the probe function is that at least one of its slots for every key is kept empty.

This is to prevent infinite looping of unsuccessful search operations. Thus, the count of filled slots for every key's probe sequence needs to be tracked such that no further insertion takes place once the respective probe sequence is left with only one empty slot.

So far, we have covered a simple form of collision-resolution policy, wherein once the home slot is filled with a key value, the subsequent key values occupy the empty slots, which are found while traversing toward the bottom of the bucket. This kind of probing for empty slots in a linear sequence is termed linear probing, which is defined as follows: p(K,i) = i . In the preceding statement, i represents the slot, which is offset by i steps down the hash table. Once the probe sequence reaches the bottom of the hash table, linear probing wraps around to start tracing from the beginning of the hash table. Thus, all the slots in the hash table are available for filling with key values before the probe sequence reaches the home slot.

Linear probing is one of the most primitive options for the resolution of collisions. However, it is one of the worst collision-resolution approaches. The main problem with linear probing is that the slot's probability of getting filled with a key value changes drastically upon insertion of every new key value into the hash table. This can be explained in detail using an illustration, as depicted in Figure 6.15 . Consider a hash table with 0 to 9 slots (or keys), whose hash function is K mod 10 . Consider five elements, which need to be inserted into the hash table in the given order: 453, 362, 396, 156, and 957. Initially, assume that each slot has an equal chance (1/10) of being a home slot, and the slot next to it has an equal chance of getting filled (due to linear probing) once its previous slot (home) gets filled. As a first step, the third slot is filled with 453, as its hash key (slot) is 3. Now, the chances of slot 4 getting filled increases to 2/10, as it can be filled either with a key value whose hash key is 3 (linear probing) or 4 (home slot). Upon second insertion, which is at slot 2, the chances of slot 4 getting filled further increases to 3/10, as now it can be filled with key values ending with 2 (linear probing), 3 (linear probing), and 4 (home slot). The chance of the remaining slots (that is, 0, 1, 5, 6, 7, 8, and 9) is still 1/10. Upon insertion of the third element, (396), at slot 6 (as home slot), the chance of slot 7 getting filled increases to 2/10, leaving other probabilities unaffected. Now, upon insertion of the fourth element (156) at slot 7 (due to linear probing), the chance of slot 8 getting filled increases to 3/10, leaving other slots' probabilities unaffected. Finally, upon insertion of the fifth element (957) at slot 8 (due to linear probing), the chance of slot 9 getting filled increases to 4/10.

[ 160 ]

Page 172

﻿

Chapter 6

Thus, the following are the resultant probabilities upon completion of all five insertions:

Figure 6.14: Probabilities obtained after insertion

Such kind of linear probing, where the slots are clustered based on their tendency to get filled up, is called primary clustering. These small clusters (at slots 4 and 9) tend to increase into a big cluster, which can further increase the discrepancy of probing:

Figure 6.15: Linear probing

One quick way of preventing primary clustering is to skip slots by a constant c instead of linearly probing by a single slot. This would modify the earlier probe function into the following: P(K,i) = ci . In the preceding function, c is a constant with a value less than the number of slots in the hash table.

The prime advantage of the former probe function is that the probe sequence traverses through all the slots of the hash table before reaching the home slot, which is not the case with the latter. Here, the traversing across slots is governed by the constant c . If c=2 , then the probe function would divide the sequence into two mutually exclusive sequences; one being an even sequence and the other odd. If the hash function returns an even home slot, and is already filled, then the traversing occurs only across all even-numbered slots before the probe sequence returns to the home slot.

[ 161 ]

Page 173

﻿

Exploring Search Options

The case when the hash function returns an odd home slot is similar. In an ideal scenario, if both the sections have a similar number of input key values, then this kind of probing has little significance.

However, if the number of input key values is different in both the sections, then the section with the higher number of key values will have more collisions and show poorer performance, whereas the other section, with fewer key values, will have a good

distribution and show better performance. Overall, the performance of the probe function decreases as the section with the higher number of collisions might dominate the declining performance.

If the constant c is relatively prime to the number of slots of the hash table, then the probe sequence will cover all the slots before it culminates at the home slot. As an example, for a table of size 10, the constant c can take the values 1, 3, 7, or 9. Similarly, for a table of size seven, any constant c lying between 1 and 6 would generate a probe sequence covering all the slots in the hash table.

Though the constant c should be able to address the issue of primary clustering, it is not in a position to completely control it. For example, when constant c assumes the value 2, the probabilities of the slots in the even and odd sequences tend to change drastically. If h(K)=4 , then the probe sequence would continue along slots 6, 8, 10, and so on. Similarly, when h(K)=6 , the probe sequence would continue along slots 8, 10, and so on. thereby directly affecting the likelihood of the next slot getting filled. This kind of high fluctuation observed in the probabilities because of interlinking between slots makes it more complex to address the issue of primary clustering.

This leads to a new form of probe sequence in which the untraced slots are randomly checked for availability. This would ensure no interlinking among slots, which is the main reason for primary clustering. Here the probe sequence should randomly select the slots for traversing. However, it is recommended to implement random slot selection, as duplication of the same probe sequence is not possible, which is inevitable for search operations. Nevertheless, pseudo-random probing can be implemented which has both the options: pseudo random selection and traceability for search. Here, the j th slot of the probe sequence is defined as (h(K) + r j ) mod M , where M is the size of the hash table, and r j is the j th slot of the random permutation of numbers between 1 and M-1 . These random permutations of numbers are stored in a vector and used for both insertion and search purposes. The probe function is written as p(K,i) = Perm[i-1] , where Perm is a vector of random numbers between 1 and M-1 .

[ 162 ]

Page 174

﻿

Chapter 6

Another form of probe function is quadratic probing, which also controls primary

clustering. The probe function defining quadratic probing is as follows: P(K,i) = c 1. i 2 + c 2. i + c 3 . In the preceding function, c 1 , c 2 , and c 3 are constants.

Quadratic probing comes with a serious disadvantage which is not applicable for many probing functions: not all the slots become a part of the probing sequence. For example, if the size of the hash table is 10, then only slots 0, 1, 4, and 9 are accessible for the quadratic function p(K,i) = i 2 . Even if the other slots are empty, they cannot be filled up as they do not become a part of probing sequence. This becomes a grave issue when some key values are left out (not inserted) of the hash table, even though some slots are empty, as those slots do not fall under the probe sequence.

In a nutshell, the right combination of hash table size and probe function will enhance the performance of insertion and search operations. If the size of the hash table is a prime number and the probe function is i 2 , then at least half of the slots in the hash table become a part of the probe sequence. Alternatively, if the size of the hash table is a power of two and the probe function is (i 2 +1)/2 , then all the slots become a part of the probe sequence. Although pseudo-random probing and quadratic probing can control primary clustering, they induce a new form of clustering known as secondary clustering. As the probe function of these methods depends only on the home slot key ( i ) instead of the key value ( K ), the probability of the slots getting filled in a probe sequence depends solely on the home slot key. If two key values are directed toward the same home slot, then the probability of only those slots which are part of the home slot's probing sequence is affected. This kind of clustering confined to a particular home slot's probing sequence, defined using a pseudo- random probe function or a quadratic probe function, is termed secondary clustering. This can be controlled if the probe function factors in the original key value ( K ) along with the home slot's key ( i ). This can be achieved using the linear probe function in which the probe sequence consists of slots separated by a constant c , and the value c is determined using a different hash function, h 2 . As a result, the modified linear probe function becomes P(K,i) = i\*h 2 (K) . This kind of two-step hashing is called double hashing. Double hashing tends to perform well when all the constants of the probe function are relatively prime to the size of the hash table ( M ). This can be achieved in two cases.

These are, when the size of the hash table ( M ) is a prime number, and the hash function h 2 returns a constant value between 1 and M-1 :

When the size of the hash table ( M ) is a power of two ( 2 m ), and the hash function h 2 returns an odd number which lies between 1 and 2 m , where m is any real number.

[ 163 ]

Page 175

﻿

Exploring Search Options

Analysis of closed hashing

This section primarily deals with the analysis of hashing. The performance of hashing mainly depends on the number of accesses made before completing an operation. The operation can be an insertion, search, or deletion. Deletion can only be implemented once the element is found in the hash table. As finding an element is a part of the search operation, the number of accesses made for a search operation is equal to the number of accesses made for a deletion operation. Similarly, to perform an insertion, the slots within a probe sequence are traversed till an empty slot is found. Also, if the key value is already present in the hash table, then it is not inserted, as it causes redundancy in the hash table. Thus, a successful search (element found in the hash table) is required for a deletion, and an unsuccessful search (element not found in the hash table) is required for an insertion. To begin with, consider an empty hash table. Then, with only one single access, insertion of the first element in its respective home slot occurs. Also, the search operation and delete operation would require only a single element access if all the elements are inserted in their respective home slots of the hash table. As the hash table starts getting filled, the probability of a new key value occupying its home slot decreases. If the new key value is hashed to an already filled home slot, then the collision resolution policy begins to search for another empty slot confined to the home slot's probe sequence. This increases the number of element accesses for performing any insertion, search, or deletion. Thus, the cost of any operation depends on the number of slots occupied within the hash table.

Let's define load factor ( α ) as the ratio of the number of slots currently filled ( N ) and the total number of slots in the hash table (size of hash table denoted as M ):

This load factor can be used analytically to obtain the cost function for an insertion operation, assuming that the probe sequence is generated using random permutation of slots. Thereby, we can safely assume that each empty slot has an equal probability of being assigned to a new key value as its home slot, and the load factor can be considered analogous to the probability of an empty slot being occupied by a new key value as its home slot. Thus, the probability of finding a home slot occupied with subsequent i probing slots, also occupied, can be defined as follows:

[ 164 ]

Page 176

﻿

Chapter 6

i

of slots in the probing sequence can be approximated as follows:

Thereby, the average cost of insertion is calculated as follows:

So far, the average cost of insertion is based on an assumption that the probe sequence is generated using random permutation of slots in the hash table. But this assumption is not always valid. Hence, the aforementioned cost represents the lower bound of the average insertion cost. Following are the true cost estimates of insertion and deletion operations using linear probing:

Insertion or unsuccessful search:

Deletion or successful search:

Thus, the growth of cost in case of insertion or unsuccessful search is faster than the growth of cost in case of deletion of a successful search. The cost defines the expected number of accesses to perform a particular operation using a hash table. On similar lines, the growth of cost in the case of linear probing is faster than the growth of cost in the case of random probing.

Deletion

An element from the hash table can only be deleted if it is successfully found during the search operation. These deletions satisfy some considerations, such as the following:

The deleted element's slot is again reusable for insertion purposes.

[ 165 ]

For larger values of N and M , the probability approximates to (N/M) . The expected number

Page 177

﻿

Exploring Search Options

The deleted element's slot does not hamper any sequential search operation. In other words, consider a probe sequence of four filled slots in which the element positioned in the second slot is deleted. The empty second slot does not

intermittently culminate any search operation, and all the subsequent slots (third and fourth) of the probe sequence will be searched.

No duplicate key values will be inserted into the deleted element's slot.

To satisfy these considerations, the deleted element's slot is specially marked, and is termed tombstone . The key features of tombstone are as follows:

It behaves as an indication of an element's deletion.

It allows search to continue as per slots in the probe sequence without any interference, if it is encountered prior to the completion of the search.

It allows for insertion of new elements if it is encountered during the insertion operation. However, prior to an insertion of a new element, the search operation is performed on the entire probe sequence (devoid of the tombstone slot) to ensure that no duplicate record is inserted. In case of multiple tombstones, the new element is inserted in the first encountered tombstone. Thus, tombstone ensures reusability.

Generally, a hash table is initially created using a set of key values, and later deletions and insertions take place, which gives rise to a set of tombstones. During the initial phase of deletions, the average length of the probe sequence increases due to tombstones, which inherently increases the distance among the elements. Upon new insertions, the count of the tombstones decreases, thereby decreasing the average distance among elements within the probe sequence. However, the decrease may not be relatively substantial. This can be explained using an example. Let us assume that the initial average path distance without any tombstones is 1.3. In other words, an average of 0.3 slots is accessed for every search operation beyond the home slot. After some deletions and insertions, the average path distances increase to 1.6 due to some tombstones. The value 1.6 may seem to be reasonable, but the relative increment of two times might seem to be a problem. This can be resolved using the following solutions:

Upon deletion, reorganization of slots within the probe sequence might help in reducing the average path distance. One crude way of performing this is to move the intermediate tombstone toward the end of the probe sequence. This can be done by simply swapping the elements (beyond the tombstone) with their previous slot such that the tombstone moves toward the end. However, this may not work for all kinds of collision resolution policies or certain probe functions.

[ 166 ]

Page 178

﻿

Chapter 6

A periodic rehashing of elements into a new hash table after a set of deletions and insertions will ensure a lower average path distance. Due to rehashing, the tombstones are removed, and the frequently accessed elements are given an opportunity to get placed in their respective home slots.

Exercises

In a jump search algorithm, the optimum number of jumps required to attain the minimum cost of comparisons is , where n is the length of the input vector. Can you derive it? Also, what is the cost of comparison for the worst case, when the number of jumps is .

Evaluate the cost of searching an unsorted and a sorted (sorting based on frequency) vector, where each element has an equal probability of being accessed during the search operation, under the following conditions:

p i = 1/n , where i is the element in the given vector p i = 1/n 2 , where i is the element in the given vector p i = 1/2 n , where i is the element in the given vector

Implement the hash function in R using the mid-square method for four-and six- digit integers.

Implement the dictionary ADT using a hash table.

Summary

The current chapter builds foundation for search arrays and hash functions. It also covers searching on sorted and un-sorted arrays. The approach utilizes vector sorting for search operations using key-value pairs. The chapter also introduces approaches such as self- organizing lists which do not utilize key values for searching rather uses expected

frequency of accessing the key values for comparison purposes. The chapter also introduces hashing concepts and covers various approaches for hashing, such as hash functions, open hashing, and closed hashing. Analysis on the computation required with the deletion operation is also covered in the final part of chapter. The next chapter will introduce tree based indexing data structures such as 2-3 trees, B-tree and B+ tree.

[ 167 ]