HASHING

Introduction

A data object called a *symbol table* is required to be defined and implemented in many applications, such as compiler/assembler writing. A symbol table is nothing but a set of pairs (name, value), where *value* represents a collection of attributes associated with the name, and the collection of attributes depends on the program element identified by the name.

For example, if a name x is used to identify an array in a program, then the attributes associated with x are the number of dimensions, lower bound and upper bound of each dimension, and element type. Therefore, a symbol table can be thought of as a linear list of pairs (name, value), and we can use a list data object for realizing a symbol table.

A symbol table is referred to or accessed frequently for adding a name, or for storing or retrieving the attributes of a name.

Therefore, accessing efficiency is a prime concern when designing a symbol table. The most common method of implementing a symbol table is to use a hash table.

Hashing is a method of directly computing the index of the table by using a suitable mathematical function called a hash function.

Note The hash function operates on the name to be stored in the symbol table, or whose attributes are to be retrieved from the symbol table.

If h is a hash function and x is a name, then h(x) gives the index of the table where x, along with its attributes, can be stored. If x is already stored in the table, then h(x) gives the index of the table where it is stored, in order to retrieve the attributes of x from the table.

There are various methods of defining a hash function. One is the division method. In this method, we take the sum of the values of the characters, divide it by the size of the table, and take the remainder. This gives us an integer value lying in the range of 0 to (n−1), if the size of the table is n.

Another method is the *mid-square method*. In this method, the identifier is first squared and then the appropriate number of bits from the middle of the square is used as the hash value. Since the middle bits of the square usually depend on all the characters in the identifier, it is expected that different identifiers will result in different values. The number of middle bits that we select depends on the table size. Therefore, if r is the number of middle bits that we are using to form the hash value, then the table size will be 2^r. So when we use this method, the table size is required to be a power of 2.

A third method is *folding*, in which the identifier is partitioned into several parts, all but the last part being of the same length. These parts are then added together to obtain the hash value.

To store the name or to add attributes of the name, we compute the hash value of the name, and place the name or attributes, as the case may be, at that place in the table whose index is the hash value of the name.

To retrieve the attribute values of the name kept in the symbol table, we apply the hash function of the name to that index of the table where we get the attributes of the name. So we find that no comparisons are required to be done; the time required for the retrieval is independent of the table size. The retrieval is possible in a constant amount of time, which will be the time taken for computing the hash function.

Therefore a hash table seems to be the best for realization of the symbol table, but there is one problem associated with the hashing, and that is collision.

Hash collision occurs when the two identifiers are mapped into the same hash value. This happens because a hash function defines a mapping from a set of valid identifiers to the set of those integers that are used as indices of the table.

Therefore we see that the domain of the mapping defined by the hash function is much larger than the range of the mapping, and hence the mapping is of a many-to-one nature. Therefore, when we implement a hash table, a suitable collision-handling mechanism is to be provided, which will be activated when there is a collision.

Collision handling involves finding an alternative location for one of the two colliding symbols. For example, if x and y are the different identifiers and h(x = h(y), x and y are the colliding symbols. If x is encountered before y, then the i^{th} entry of the table will be used for accommodating the symbol x, but later on when y comes, there is a hash collision. Therefore we have to find a suitable alternative location either for x or y. This means we can either accommodate y in that location, or we can move x to that location and place y in the i^{th} location of the table.

Various methods are available to obtain an alternative location to handle the collision. They differ from each other in the way in which a search is made for an alternative location. The following are commonly used collision-handling techniques:

Linear Probing or Linear Open Addressing

In this method, if for an identifier x, h(x) = i, and if the i^{th} location is already occupied, we search for a location close to the i^{th} location by doing a linear search, starting from the $(i+1)^{th}$ location to accommodate x. This means we start from the $(i+1)^{th}$ location and do the linear search until we get an empty location; once we get an empty location we accommodate x there.

Rehashing

In *rehashing* we find an alternative empty location by modifying the hash function and applying the modified hash function to the colliding symbol. For example, if x is the symbol and h(x) = i, and if the i^{th} location is already occupied, then we modify the hash function h to h_1, and find out $h_1(x)$, if $h_1(x)$ = j. If the j^{th} location is empty, then we accommodate x in the j^{th} location. Otherwise, we once again modify h_1 to some h_2 and repeat the process until the collision is handled. Once the collision is handled, we revert to the original hash function before considering the next symbol.

Overflow chaining

Overflow chaining is a method of implementing a hash table in which the collisions are handled automatically. In this method, we use two tables: a symbol table to accommodate identifiers and their attributes, and a *hash table*, which is an array of pointers pointing to symbol table entries. Each symbol table entry is made of three fields: the first for holding the identifier, the second for holding the attributes, and the third for holding the link or pointer that can be made to point to any symbol table entry. The insertions into the symbol table are done as follows:

If x is the symbol to be inserted, it will be added to the next available entry of the symbol table. The hash value of x is then computed. If h(x) = i, then the i^{th} hash table pointer is made to point to the symbol table entry in which x is stored, if the i^{th} hash table pointer does not point to any symbol table entry. If the i^{th} hash table pointer is already pointing to some symbol table entry, then the link field of the symbol table entry containing x is made to point to that symbol table entry to which the i^{th} hash table pointer is pointing to, and the i^{th} hash table pointer is made to point to the symbol entry containing x. This is equivalent to building a linked list on the i^{th} index of the hash table. The retrieval of attributes is done as follows:

If x is a symbol, then we obtain h(x), use this value as the index of the hash table, and traverse the list built on this index to get that entry which contains x. A typical hash table implemented using this technique is shown here.

The symbols to b stored are x_1,y_1,z_1,x_2,y_2,z_2. The hash function that we use is h(symbol) = (value of first letter of the symbol) mod n, where n is the size of table.

if $h(x_1) = I$ $h(y_1) = j$ $h(z_1) = k$ $h(x_2) = I$ $h(y_2) = j$ $h(z_2) = k$

SEARCHING TECHNIQUES: LINEAR OR SEQUENTIAL SEARCH

Introduction

There are many applications requiring a search for a particular element. Searching refers to finding out whether a particular element is present in the list. The method that we use for this depends on how the elements of the list are organized. If the list is an unordered list, then we use linear or sequential search, whereas if the list is an ordered list, then we use binary search.

The search proceeds by sequentially comparing the key with elements in the list, and continues until either we find a match or the end of the list is encountered. If we find a match, the search terminates successfully by returning the index of the element in the list which has matched. If the end of the list is encountered without a match, the search terminates unsuccessfully.

Program

```
#include <stdio.h>
#define MAX 10
void lsearch(int list[],int n,int element)
{
   int i, flag = 0;
   for(i=0;i<n;i++)
   if( list[i] == element)
   {
      printf(" The element whose value is %d is present at position %d
in list\n",element,i);
      flag =1;
      break;
   }
   if( flag == 0)
printf("The element whose value is %d is not present in the list\n",
             element);
}

void readlist(int list[],int n)
{
   int i;
   printf("Enter the elements\n");
   for(i=0;i<n;i++)
       scanf("%d",&list[i]);
}

void printlist(int list[],int n)
{
   int i;
   printf("The elements of the list are: \n");
   for(i=0;i<n;i++)
       printf("%d\t",list[i]);
}
void main()
{
    int list[MAX], n, element;
   printf("Enter the number of elements in the list max = 10\n");
   scanf("%d",&n);
   readlist(list,n);
   printf("\nThe list before sorting is:\n");
   printlist(list,n);
   printf("\nEnter the element to be searched\n");
   scanf("%d",&element);
   lsearch(list,n,element);
}
```

Example

Input

Enter the number of elements in the list, max = 10

```
10
```

Enter the elements

```
23
1
45
67
90
100
432
15
77
55
```

Output

The list before sorting is:

The elements of the list are:

```
23 1 45 67 90 100 432 15 77 55
```

Enter the element to be searched

```
100
```

The element whose value is 100 is present at position 5 in list

Input

Enter the number of elements in the list max = 10

```
10
```

Enter the elements

```
23
1
45
67
90
101
23
56
44
22
```

Output

The list before sorting is:

The elements of the list are:

```
23 1 45 67 90 101 23 56 44 22
```

Enter the element to be searched 100

Introduction

Heapsort is a sorting technique that sorts a contiguous list of length *n* with O(*n* $\log_2$ (*n*)) comparisons and movement of entries, even in the worst case. Hence it achieves the worst-case bounds better than those of quick sort, and for the contiguous list, it is better than merge sort, since it needs only a small and constant amount of space apart from the list being sorted.

Heapsort proceeds in two phases. First, all the entries in the list are arranged to satisfy the heap property, and then the top of the heap is removed and another entry is promoted to take its place repeatedly. Therefore, we need a function that builds an initial heap to arrange all the entries in the list to satisfy the heap property. The function that builds an initial heap uses a function that adjusts the ith entry in the list, whose entries at 2i and 2i + 1 positions already satisfy the heap property in such a manner that the entry at the ith position in the list will also satisfy the heap property.

Program

```
#include <stdio.h>
#define MAX 10
void swap(int *x,int *y)
{
   int temp;
   temp = *x;
   *x = *y;
   *y = temp;
}

void adjust( int list[],int i, int n)
{
   int j,k,flag;
   k = list[i];
   flag = 1;
   j = 2 * i;
   while(j <= n && flag)
   {
      if(j < n && list[j] < list[j+1])
      j++;
      if( k >= list[j])
               flag =0;
      else
      {
         list[j/2] = list[j];
         j = j *2;
      }
   }
   list [j/2] = k;
}

void build_initial_heap( int list[], int n)
{
   int i;
   for(i=(n/2);i>=0;i-)
        adjust(list,i,n-1);
}

void heapsort(int list[],int n)
{
   int i;
   build_initial_heap(list,n);
   for(i=(n-2); i>=0;i-)
   {
      swap(&list[0],&list[i+1]);
      adjust(list,0,i);
   }
}
```

```
void readlist(int list[],int n)
{
   int i;
   printf("Enter the elements\n");
   for(i=0;i<n;i++)
       scanf("%d",&list[i]);
}

void printlist(int list[],int n)
{
   int i;
   printf("The elements of the list are: \n");
   for(i=0;i<n;i++)
       printf("%d\t",list[i]);
}

void main()
{
   int list[MAX], n;
   printf("Enter the number of elements in the list max = 10\n");
   scanf("%d",&n);
   readlist(list,n);
   printf("The list before sorting is:\n");
   printlist(list,n);
   heapsort(list,n);
   printf("The list after sorting is:\n");
   printlist(list,n);
}
```

Example

Input

Enter the number of elements in the list, max = 10

```
10
```

Enter the elements

```
56
1
34
42
90
66
87
12
21
11
```

Output

The list before sorting is:

The elements of the list are:

```
56 1 34 42 90 66 87 12 21 11
```

The list after sorting is:

The elements of the list are:

```
1 11 12 21 34 42 56 66 87 90
```

Explanation

In each pass of the `while` loop in the function `adjust(x, i, n)`, the position `i` is doubled, so the number of passes cannot exceed log(*n*/i). Therefore, the computation time of adjust is O(log *n*/i).

The function `build_initial_heap` calls the adjust procedure n/2 for values ranging from n1/2 to 0. Hence the total number of iterations will be:

log(n)+ log(n/2)+……+log(n/n/2)
n/2
=Σ log(n/i)
i=1

=n/2log(n)-log(!n/2)

This turns out to be some constant time n. So the computation time of build_initial_heap is O(n). The heapsort function calls the adjust (x,1, i) (*n*−1) times. So the total number of iterations made in the heapsort will be

log(i/1)

=n-1

log(i)

i=1

=log(1)+ log(2)+…..+log(n-1)

which turns out to be approximately n log(n). So the computing time of heapsort is O(n log(n)) + O(n). The only additional space needed by heapsort is the space for one record to carry out the exchange.

POLYNOMIAL REPRESENTATION

One of the problems that a linked list can deal with is manipulation of symbolic polynomials. By symbolic, we mean that a polynomial is viewed as a list of coefficients and exponents. For example, the polynomial

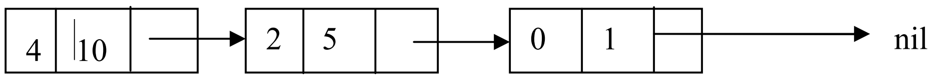
$3x^2+2x+4$,

can be viewed as list of the following pairs

(3,2),(2,1),(4,0)

Exp	Coef	link

A polynomial $10x^4 + 5x^2 + 1$ can be represented as shown here:



The procedure to add these two polynomials using the linked list is in the following program.

Program

```
# include <stdio.h>
# include <stdlib.h>
struct pnode
   {
      int exp;
      double coeff;
      struct pnode *link;
   };

struct pnode *insert(struct pnode *, int,double);
void printlist ( struct pnode * );
struct pnode *polyadd(struct pnode *, struct pnode *);
struct pnode *sortlist(struct pnode *);

struct pnode *insert(struct pnode *p, int e,double c)
{
   struct pnode *temp;
   if(p==NULL)
   {
      p=(struct pnode *)malloc(sizeof(struct pnode));
      if(p==NULL)
      {
         printf("Error\n");
         exit(0);
      }
      p-> exp = e;
      p->coeff = c;
      p-> link = NULL;
   }
   else
   {
      temp = p;
      while (temp-> link!= NULL)
      temp = temp-> link;
      temp-> link = (struct pnode *)malloc(sizeof(struct pnode));
      if(temp -> link == NULL)
      {
   printf("Error\n");
         exit(0);
      }
    temp = temp-> link;
      temp-> exp = e;
      temp->coeff = c;
      temp-> link = NULL;
   }
   return (p);
}

/* a function to sort a list */
struct pnode *sortlist(struct pnode *p)
{
   struct pnode *temp1,*temp2,*max,*prev,*q;
   q = NULL;
   while(p != NULL)
   {
      prev = NULL;
      max = temp1 = p;
      temp2 = p -> link;
      while ( temp2 != NULL )
      {
         if(max -> exp < temp2 -> exp)
              {
         max = temp2;
                prev = temp1;
              }
              temp1 = temp2;
              temp2 = temp2-> link;
            }
      if(prev == NULL)
            p = max -> link;
            else
            prev -> link = max -> link;
            max -> link = NULL;
            if( q == NULL)
            q = max; /* moves the node with highest data value in the list
pointed to by p to the list
   pointed to by q as a first node*/
         else
         {
            temp1 = q;
            /* traverses the list pointed to by q to get pointer to its
last node */
         while( temp1 -> link != NULL)
         temp1 = temp1 -> link;
         temp1 -> link = max; /* moves the node with highest data value
in the list pointed to
by p to the list pointed to by q at the end of list pointed by
q*/

      }
   }
   return (q);
}
/* A function to add two polynomials */
struct pnode *polyadd(struct pnode *p, struct pnode *q)
{
   struct pnode *r = NULL;
   int e;
   double c;
   while((p!=NULL) && (q != NULL))
            {
               if(p->exp > q->exp)
                {
                     r = insert(r,p->exp,p->coeff);
                     p = p->link;
                }
                else
                   if(p->exp < q->exp)
                 {
                     r = insert(r,q->exp,q->coeff);
                     q = q->link;
                  }
                  else
{
   c = p->coeff + q->coeff;
   e = q->exp;
   r = insert( r, e,c);
   p = p->link;
   q = q->link;
         }
while(p != NULL)
         {
               r = insert( r, p->exp,p->coeff);
               p = p->link;
         }
         while(q!=NULL)
         {
                  r = insert( r, q->exp,q->coeff);
                  q = q->link;
         }
return(r);
}

void printlist ( struct pnode *p )
{
   printf("The polynomial is\n");
   while (p!= NULL)
        {
   printf("%d %lf\t",p-> exp,p->coeff);
            p = p-> link;
      }
}
void main()
{
      int e;
```

```
         int n,i;
         double c;
         struct pnode *poly1 = NULL ;
         struct pnode *poly2=NULL;
         struct pnode *result;
         printf("Enter the terms in the polynomial1 \n");
         scanf("%d",&n);
         i=1;
         while ( n-- > 0 )
         {
      printf( "Enter the exponent and coefficient of the term number
%d\n",i);
               scanf("%d %lf",&e,&c);
               poly1 = insert ( poly1,e,c);
         }
        printf("Enter the terms in the polynomial2 \n");
         scanf("%d",&n);
         i=1;
         while ( n-- > 0 )
         {
      printf( "Enter the exponent and coefficient of the term number
%d\n",i);
               scanf("%d %lf",&e,&c);
               poly2 = insert ( poly2,e,c);
         }
         poly1 = sortlist(poly1);
         poly2 = sortlist(poly2);
         printf("The polynomial 1 is\n");
         printlist ( poly1 );
         printf("The polynomial 2 is\n");
         printlist ( poly2 );
        result = polyadd(poly1,poly2);
         printf("The result of addition is\n");
         printlist ( result );
   }
```

Explanation

1. If the polynomials to be added have n and m terms, respectively, then the linked list representation of these polynomials contains m and n terms, respectively.
2. Since `polyadd` traverses each of these lists, sequentially, the maximum number of iterations that `polyadd` will make will not be more than $m + n$. So the computation time of `polyadd` is $O(m + n)$.

COUNTING THE NUMBER OF NODES OF A LINKED LIST

Counting the number of nodes of a singly linked list requires maintaining a counter that is initialized to 0 and incremented by 1 each time a node is encountered in the process of traversing a list from the start.

Here is a complete program that counts the number of nodes in a singly linked chain p, where p is a pointer to the first node in the list.

Program

```
# include <stdio.h>
# include <stdlib.h>
struct node
{
int data;
struct node *link;
};
struct node *insert(struct node *, int);
int nodecount(struct node*);
void printlist ( struct node * );

struct node *insert(struct node *p, int n)
{
   struct node *temp;
   if(p==NULL)
   {
      p=(struct node *)malloc(sizeof(struct node));
      if(p==NULL)
      {
   printf("Error\n");
          exit(0);
      }
      p-> data = n;
      p-> link = NULL;
   }
   else
   {
      temp = p;
      while (temp-> link!= NULL)
     temp = temp-> link;
      temp-> link = (struct node *)malloc(sizeof(struct node));
      if(temp -> link == NULL)
      {
   printf("Error\n");
         exit(0);
      }
     temp = temp-> link;
      temp-> data = n;
      temp-> link = NULL;
      }
      return (p);
}

void printlist ( struct node *p )
{
      printf("The data values in the list are\n");
      while (p!= NULL)
      {
   printf("%d\t",p-> data);
         p = p-> link;
      }
}
```

```
/* A function to count the number of nodes in a singly linked list */
int nodecount (struct node *p )
{
   int count=0;
   while (p != NULL)
   {
      count ++;
      p = p->link;
   }
      return(count);
}

void main()
{
   int n;
   int x;
   struct node *start = NULL ;
   printf("Enter the nodes to be created \n");
   scanf("%d",&n);
   while ( n-- > 0 )
   {
printf( "Enter the data values to be placed in a node\n");
      scanf("%d",&x);
      start = insert ( start,x);
   }
   printf("The created list is\n");
   printlist ( start );
   n = nodecount(start);
   printf("The number of nodes in a list are: %d\n",n);

}
```

PROBLEM: IMPLEMENTATION OF A HASH SEARCH

Write a program that takes strings as inputs and stores them in a hash table. It should then ask the user for strings to be searched in the hash table. Use shift- folding as the hashing function and chaining for overflow handling.

Program

```
#include <stdio.h>
#include <string.h>
#include <malloc.h>

#define MAXLEN 80
#define HASHSIZE 23              // some prime val.
#define SHIFTBY 3                // each group size in hashing.

typedef struct node node;
typedef char *type;
typedef node *hashtable[HASHSIZE];

struct node {
   int val;
   char *key;
   node *next;
};
```

```c
int hGetIndex(char *key) {
   /*
    * returns index into hashtable applying hash function.
    * uses shift-folding followed by mod function for hashing.
    */
   int i, n, finaln=0;
   char *keyptr;

   for(keyptr=key; *keyptr; finaln+=n)
       for(i=0, n=0; i<SHIFTBY && *keyptr; ++i, ++keyptr)
              n = n*10 + *keyptr;
   finaln %= HASHSIZE;

   return finaln;
}

void hInsert(hashtable h, char *key, int val) {
    /*
     * insert s in hashtable h.
     * use shift-folding followed by mod function for hashing.
     * does NOT check for duplicate insertion.
     */
   node *ptr = (node *)malloc(sizeof(node));
   int index = hGetIndex(key);

   ptr->key = strdup(key);
   ptr->val = val;
   ptr->next = h[index];

   h[index] = ptr;
   printf("h[%d] = %s.\n", index, key);
}

int hGetVal(hashtable h, char *key) {
   /*
    * returns val corresponding to key if present in h else -1.
    */
   node *ptr;

   for(ptr=h[hGetIndex(key)]; ptr && strcmp(ptr->key, key); ptr=ptr->next)
      ;
   if(ptr)
       return ptr->val;
   return -1;
}

void printHash(hashtable h) {
    /*
     * print the hashtable rowwise.
     */
   int i;
   node *ptr;

   for(i=0; i<HASHSIZE; ++i) {
       printf("%d: ", i);
       for(ptr=h[i]; ptr; ptr=ptr->next)
              printf("%s=%d ", ptr->key, ptr->val);
       printf("\n");
    }
}

int main() {
    char s[MAXLEN];
    int i = 0;
    hashtable h = {"abc"};

    printf("Enter the string to be hashed: ");
    gets(s);

    while(*s) {
       hInsert(h, s, i++);
       printf("Enter the string to be hashed(enter to end): ");
       gets(s);
    }
    printf("Enter the string to be searched: ");
    gets(s);

    while(*s) {
       printf("%s was inserted at number %d.\n", s, hGetVal(h, s));
       printf("\nEnter the string to be searched(enter to end): ");
       gets(s);
    }
    //printHash(h);

    return 0;
}
```

Explanation

1. The hash table is maintained as an array of lists. Each list is either empty or contains nodes containing strings that map to the index in the hash table, after application of the hashing function. The string in the node is called the key. Each node also stores an integer that is the number at which the string was inserted in the hash table. Thus, each node contains a (key, value) pair. In a realistic situation, a value can be anything that has a key associated with it.
2. The program contains two loops. In the first, it asks the user to enter a series of strings and calls `hInsert()` to insert the strings in the hash table. The second loop asks the user to enter a string and returns the number at which it was inserted. An insertion number of −1 indicates that the string is not present in the hash table. This is done by using the function `hGetVal()`. Both these functions make use of the hashing function `hGetIndex()`, which, given a string, returns its hashing index. It folds the string into a pattern of *m* characters (perhaps except the last), and forms an integer out of each *m* characters. It then adds all these integers to get another number. This is then divided by the size of the hash table array to get the remainder as an index into the hash table. `hInsert()` adds this new string and its insertion sequence to a node and this node is added to the start of the list in the index. `hGetVal()` searches the list at this index for the input string. If it finds such a string, its insertion sequence is returned, otherwise it returns −1.
3. Since the complexity of insertion of a node in the list is O(1), the complexity of `hInsert()` is the complexity of the hashing function. The complexity of the hashing function is O(*p*) where p is the average length of the string. Thus the complexity of `hInsert()` is O(*p*). The complexity

of `hGetVal()` is O(p+q) where q is the average number of nodes in each list. Chaining involves a linear search.

Points to Remember

1. The complexity of insertion in the hash table is decided by the hashing function if simple chaining is used for overflow handling.
2. The complexity of searching is decided by both the hashing function and the overflow handling technique.
3. An ideal hash function maps every input string to a different index and thus has zero collisions. Assuming that the complexity of a hash function is O(1), the insertions and searching into an ideal hash table are O(1).
4. Hash tables are used in compilers for symbol-table management. Hash tables have numerous other applications as well.
5. Different overflow handling techniques such as linear probing, quadratic probing, random probing, rehashing, etc., are in use depending on the application requirement.