Hashing, File I/O

ESC101: Fundamentals of Computing

Nisheeth

Hashing for Very Fast Search

- Hashing is a method to search an element in an array in constant time
- "Constant time" also denoted as O(1) – means time taken does not depend on number of elements N in the array (unlike like binary/brute force search)
- Since we can search in constant time, can also update/delete in constant time
- Basic idea: Use a "hash table" to store the elements
- The hash table is just like an array

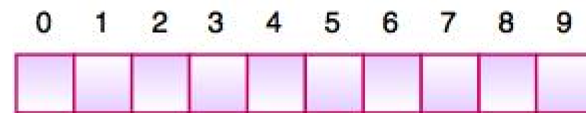


Fig. Hash Table

- Index of each element to be stored is calculated using the element's value
- To search the element, compute its index and directly find it at that index
- This can be done in constant time (if index can be found in constant time) ☺

Hash Function

- Index is computed using a hash function
- Hash function uses the element's value to compute its index
- An example of a simple hash function is the modulo operator

$$index = value \ \% \ number_of_slots$$

Value	Index = Value % No. of Slots
26	26 % 10 = 6
70	70 % 10 = 0
18	18 % 10 = 8
31	31 % 10 = 1
54	54 % 10 = 4
93	93 % 10 = 3

0	1	2	3	4	5	6	7	8	9
70	31		93	54		26		18	

Fig. Hash Table

To search for an element, say 93, we simply apply
the hash function again: 93%10 = 3
and can find the index of this element in constant time ☺

A potential problem - collisions

- What if more than multiple elements get mapped to the same index?
- Yes, a very real problem.

A potential problem - collisions

- What if more than multiple elements get mapped to the same index?
- Yes, a very real problem.
- Consider the previous hash table

0	1	2	3	4	5	6	7	8	9
70	31		93	54		26		18	

Fig. Hash Table

- Suppose we wish to insert 60. Its index = 60%10 = 0 => clash with 70
- Some hash functions are good in the sense that the indices they generate are uniformly distributed (so less collision - desirable for good hash functions)
- Despite that, we may still have collisions and we need to handle that

भारतीय प्रौद्योगिकी संस्थान कानपुर INDIAN INSTITUTE OF TECHNOLOGY KANPUR

Linear Probing

0	1	2	3	4	5	6	7	8	9
70	31		93	54		26		18	

Fig. Hash Table

- Linear Probing is a simple technique to handle collisions
- The idea: Keep searching for the "next available" free index
- Assume the first index P that we get is not free. Then compute

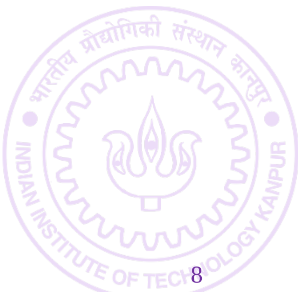
$$P = (P+1) \,\%\, number_of_slots$$

- If the new index P is free, store the element there, else repeat the above
- Suppose we wish to insert 60 in this table, 60%10 = 0, but 0 if not free
- Try P = (P+1)%10 = (0+1)%10 = 1, but index 1 is also not free (31 there)
- Let's try P = (P+1)%10 = (1+1)%10 = 2. Index 2 is free. Store 60 at that index
- When searching for 60, we won't find it in first attempt but in third attempt

Hashing: Some final thoughts..

- A very very useful technique
- We have only scratched the surface – the basic idea of hashing
- More advanced hashing methods exist
  - Better methods to avoid collisions
  - Better and cheap to compute hashing functions
- Discussion of these is beyond the scope of ESC101

File Input/Output



Files

- What is a file?
  - Collection of bytes stored on secondary storage like hard disks (not RAM which is primary storage).
- Any *addressable* part of the file system in an operating system can be a file.
  - includes such strange things as /dev/null (nothing), /dev/usb (USB port), /dev/audio (speakers), and of course, files that a user creates (/home/don/input.txt, /home/don/Esc101/lab12.c)

File Access

- 3 files are always connected to a C program :
  - stdin : the standard input, from where scanf, getchar(), gets() etc. read input from
  - stdout : the standard output, to where printf(), putchar(), puts() etc. output to.
  - stderr : standard error console.

File handling in C

1. Open the file for reading/writing etc.: fopen
   - return a *file pointer*
   - pointer points to an internal structure containing information about the file:
     - location of a file
     - the current position being read in the file, etc.

   FILE* fopen (char *name, char *mode)

2. Read/Write to the file

   int fscanf(FILE *fp, char *format, …)

   int fprintf(FILE *fp, char *format, …)

   int fputs(const char* str, FILE *fp)

3. Close the File.

   int fclose(FILE *fp)

Compared to scanf and printf – a new (first) argument fp is added

Opening Files

FILE* fopen (char *name, char *mode)

- The first argument is the name of the file
  - can be given in short form (e.g. “inputfile”) or the full path name (e.g. “/home/don/inputfile”)
- The second argument is the mode in which we want to open the file. Common modes include:
  - “r” : read-only. Any write to the file will fail. File must exist.
  - “w” : write. The first write happens at the beginning of the file, by default. Thus, may overwrite the current content. A new file is created if it does not exist.
  - “a” : append. The first write is to the end of the current content. File is created if it does not exist.

Opening Files

- If successful, fopen returns a *file pointer* – this is later used for fprintf, fscanf etc.
- If unsuccessful, fopen returns a NULL.
- It is a good idea to check for errors (e.g. Opening a file on a CDROM using “w” mode etc.)

Closing Files

- An open file must be closed after last use
  - allows reuse of FILE* resources
  - flushing of ***buffered*** data (to actually write!)

File I/O: Example

- Write a program that will take two filenames, and print contents to the standard output. The contents of the first file should be printed first, and then the contents of the second.
- The algorithm:

  1. Read the file names.
  2. Open file 1. If open failed, we exit
  3. Print the contents of file 1 to stdout
  4. Close file 1
  5. Open file 2. If open failed, we exit
  6. Print the contents of file 2 to stdout
  7. Close file 2

```
int main()
{
  FILE *fp;  char filename1[128], filename2[128];
  scanf("%s", filename1);
  scanf("%s", filename2);
  fp = fopen( filename1, "r" );
  if(fp == NULL) {
    fprintf(stderr, "Opening File %s failed\n", filename1);
    return -1;
  }
  copy_file(fp, stdout);
  fclose(fp);
  fp = fopen( filename2, "r" );
  if (fp == NULL) {
    fprintf(stderr, "Opening File %s failed\n", filename2);
    return -1;
  }
  copy_file (fp, stdout);
  fclose(fp);
  return 0;
}
```

The Program: copy_file

```
void copy_file(FILE *fromfp, FILE *tofp)
{
  char ch;

  while ( !feof ( fromfp ) ) {
    fscanf  ( fromfp, "%c", &ch );
    fprintf ( tofp, "%c", ch );
  }
}
```

Some other file handling functions

- `int feof ( FILE* fp );`
  - Checks whether the fp has reached EOF – that is, the EOF chracter has been encountered. If EOF is found, it returns nonzero. Otherwise, returns 0.
- `int ferror ( FILE *fp );`
  - Checks whether the error indicator has been set for fp. (for example, write errors to the file.)

Some other file handling functions

- `int fseek(FILE *fp, long int offset, int origin);`
  - To set the current position associated with fp, to a new position = origin + offset.
  - Origin can be:
    - SEEK_SET: beginning of file
    - SEEK_CURR: current position of file pointer
    - SEEK_END: End of file
  - Offset is the number of bytes.
- `int ftell(FILE *fp)`
  - Returns the current value of the position indicator of the stream.

Opening Files: More modes

- There are other modes for opening files, as well.
  - "r+" : open a file for read and update. The file must be present.
  - "w+" : write/read. Create an empty file or overwrite an existing one.
  - "a+" : append/read. File is created if it doesn't exist. The file position for reading is at the beginning, but output is appended to the end.

File I/O example

```
#include <stdio.h>
int main () {
        FILE * fp = fopen("file.txt","w+");
        fputs("This is tutorialspoint.com", fp);
        fseek( fp, 7, SEEK_SET );
        fputs(" C Programming Language", fp);
        fclose(fp);

        int c;
        fp = fopen("file.txt","r");
        while(1)  {
                c = fgetc(fp);
                if( feof(fp) ) break;
                printf("%c", c);
        }
        fclose(fp);
        return 0;
}
```

This is C Programming Language