DATA STRUCTURES AND ALGORITHMS

Sorting algorithms

External sorting, Search

Summary of the previous lecture

- Fast sorting algorithms
  - Quick sort
  - Heap sort
  - Radix sort
- Running time of these algorithms in average is O(n log n).

External sorting

Earlier analyzed basic data structures and algorithms operate on data stored in main memory.
Some applications require that large amounts of information be stored and processed—so much information that it cannot all fit into main memory.

In that case, the information **must reside on disk and be brought into main memory selectively for processing**.

Primary versus Secondary Storage

Computer storage devices are typically classified into:

- **primary** or main memory (Random Access Memory)
- **secondary** or peripheral storage (hard disk drives, removeable "flash" drives, floppy disks, CDs, DVDs, and tape drives).

Primary memory also includes registers, cache, and video memories.

Access a byte of storage from a HDD is around 5 - 9 ms while typical access time from standard personal computer RAM is about 5 - 10 nanoseconds.

The number of disk accesses must be Minimized !

External sorting

Consider the problem of sorting collections of records too large to fit in main memory. Because the records must reside in peripheral or external memory, such sorting methods are called **external sorts**.

This is in contrast to the **internal sorts** which assume that the records to be sorted are stored in main memory.

External sorting

When a collection of records is too large to fit in main memory, the only practical way to sort it is to read some records from disk, do some rearranging, then write them back to disk.
This process is repeated until the file is sorted, with each record read perhaps many times.

Given the high cost of disk I/O, it should come as no surprise that the primary **goal of an external sorting algorithm is to minimize the amount of information that must be read from or written to disk**.

Block size

Sector sizes of the disc are typically a power of two, in the range 512 to 16K bytes, depending on the operating system and the size and speed of the disk drive. The block size used for external sorting algorithms should be equal to or a multiple of the sector size.

Under this model, a sorting algorithm reads a **block of data** into a buffer in main memory, performs some processing on it, and at some future time writes it back to disk.

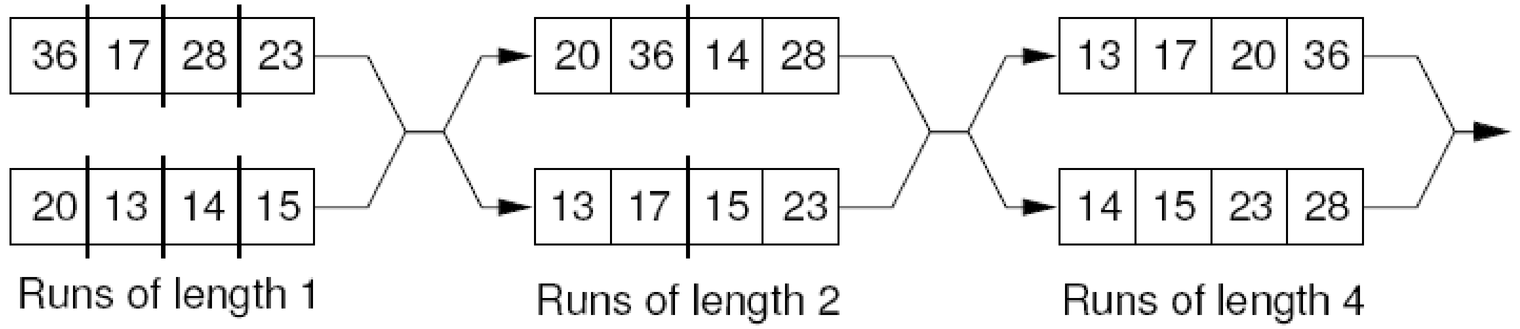
Records contained in a single block can be sorted by an internal sorting algorithm such as Quicksort in less time.

External sort

Our approach to external sorting is derived from the Mergesort algorithm.
The simplest form of external Mergesort performs a series of sequential passes over the records, merging larger and larger sublists on each pass:
the first pass merges sublists of size 1 into sublists of size 2;
the second pass merges the sublists of size 2 into sublists of size 4; and so on.

A sorted sublist is called a run.
Thus, each pass is merging pairs of runs to form longer runs.
Each pass copies the contents of the file to another file.

External Mergesort algorithm



External Mergesort algorithm

1. Split the original file into two equal-sized run files.
2. Read one block from each run file into input buffers.
3. Take the first record from each input buffer, and write a sublist of length two to an output buffer in sorted order.
4. Take the next record from each input buffer, and write a sublist of length two to a second output buffer in sorted order.
5. Repeat until finished, alternating output between the two output run buffers. Whenever the end of an input block is reached, read the next block from the appropriate input file. When an output buffer is full, write it to the appropriate output file.
6. Repeat steps 2 through 5, using the original output files as input files. On the second pass, the first two records of each input run file are already in sorted order. Thus, these two runs may be merged and output as a single run of four elements.
7. Each pass through the run files provides larger and larger sublists until only one sublist remains.

Searching

Searching is the most frequently performed of all computing tasks.

Abstract definition
Search is a process to determine if an element with a particular value is a member of a particular set.

Common definition
Search is a process how to find the record within a collection of records that has a particular key value, or those records in a collection whose key values meet some criterion such as falling within a range of values.

Searching

We can define searching formally as follows. Suppose k_1, k_2, ... k_n are distinct keys, and that we have a collection **L** of n records of the form

$$(k_1, I_1), (k_2, I_2), ..., (k_n, I_n)$$

where I_j is information associated with key k_j for 1<= j <= n. Given a particular key value **K**, the search problem is to locate the record (k_j ; I_j) in **L** such that k_j = K (if one exists). Searching is a systematic method for locating the record (or records) with key value k_j = K.

A **successful search** is one in which a record with key k_j = K is found.

An **unsuccessful search** is one in which no record with k_j = K is found (and no such record exists).

An **exact-match** query is a search for the record whose key value matches a specified key value.

A **range query** is a search for all records whose key value falls within a specified range of key values.

Searching

Search algorithms falls into three general approaches:

1. Sequential and list methods.
2. Direct access by key value (hashing).
3. Tree indexing methods.

Running time

How many comparisons does linear search do on average? A major consideration is whether **K** is in list **L** at all.
We can simplify our analysis by ignoring everything about the input except the position of **K** if it is found in **L**. Thus, we have **n + 1** distinct possible events: that **K** is in one of positions 0 to n - 1 in **L** (each with its own probability), or that it is not in **L** at all.

We can express the probability that **K** is not in **L** as:

$$\mathbf{P}(K \notin \mathbf{L}) = 1 - \sum_{i=1}^{n} \mathbf{P}(K = \mathbf{L}[i])$$

where P(x) is the probability of event x.

Let p_i be the probability that K is in position i of $\mathbf{L}$. When K is not in $\mathbf{L}$, sequential search will require n comparisons. Let p_0 be the probability that K is not in $\mathbf{L}$. Then the average cost $\mathbf{T}(n)$ will be

$$\mathbf{T}(n) = np_0 + \sum_{i=1}^{n} ip_i.$$

What happens to the equation if we assume all the p_i's are equal (except p_0)?

$$\begin{aligned}
\mathbf{T}(n) &= p_0 n + \sum_{i=1}^{n} ip \\
&= p_0 n + p\sum_{i=1}^{n} i \\
&= p_0 n + p\frac{n(n+1)}{2} \\
&= p_0 n + \frac{1-p_0}{n}\frac{n(n+1)}{2} \\
&= \frac{n+1+p_0(n-1)}{2}
\end{aligned}$$

Depending on the value of p_0, $\frac{n+1}{2} \leq \mathbf{T}(n) \leq n$.

Sequential search

```
#include <stdio.h>
int main(void)
{     int array[10]; int key;
      array[0]=20; array[1]=40; array[2]=100; array[3]=80; array[4]=10;
      array[5]=60; array[6]=50; array[7]=90; array[8]=30; array[9]=70;
      printf("Enter the number you want to find (from 10 to 100):");
            scanf("%d", &key);
      int flag = 0;                          // set flag to off

      for (int i=0; i<10; i++)       // start to loop through the array
     {    if ( array[i] == key)       // if match is found
           {       flag = 1;              // turn flag on
                   break ;                // break out of for loop
            }       }
      if (flag)
          printf ("Your number is at subscript position %d\n",  i);
      else
          printf ("Sorry, I could not find your number in this array\n");
      }
     return 0;}
```