Universität Innsbruck - Institut für Informatik
Prof. Clemens Ballarin, Robert Binna, Friedrich Neurauter, Francois Scharffe and Sarah Winkler

17 June 2008

Proseminar Algorithmen und Datenstrukturen

Exercise Sheet 12

Exercise 1 (Combined Sorting)

In the last exercise sheets you implemented several different sorting algorithms. All these sorting problems had one thing in common, they used the same algorithm for the whole problem, independent of the problem size. Another strategy for sorting is to use a divide and conquer algorithm for large problems and when the problem gets small enough the algorithm switches to more suitable one. The decision what algorithm is used is made during the sorting process after a certain subproblem size was reached.

a) Extend the sorting framework from the previous exercises with the ability to measure the runtime of the different algorithms. (Hint use gettimeofday for measuring time).

b) Try different sorting algorithms for different problem sizes and measure what algorithm is suitable for which problem size. Give an explanation of your results.
SOLUTION: After measuring the execution time with 20000 tests on each input array and between 100 and 1000 different input arrays. It became clear that the threshold between quick sort and insert sort will be at $n = 11$.

c) Use the results from above to implement a combined sorting algorithm that uses a divide and conquer based algorithm for large problem sizes and a simpler, more suitable algorithm for smaller subproblems. Set the threshold for the problem size according to your results from the exercise above. Integrate your algorithm into the existing framework from the previous exercises.

d) Compare the execution time of the new combined sorting algorithm with the other “Text in standalone” algorithms.

Exercise 2 (Choose the right sorting algorithm)

Given are several scenarios where sorting algorithms are applied. Choose an appropriate sorting algorithm and explain why the algorithm was chosen:

a) Given is a list of people sorted by their year of birth. Sort them by their resting pulse rate. (Hint: the higher the age the lower the resting puls rate)
SOLUTION: Because the list is sorted by year of birth it is almost sorted by puls. Therefore insertion sort is a good algorithm for this problem.

b) Given is a cryptographic algorithm that facilitates a sorting algorithm as part of its implementation. Which algorithms can prevent timing attacks?
SOLUTION: A good solution would be heapsort because it provides stable runtime behaviour.

c) Given is an alphabetic sorted list of students of the university of Innsbruck. Find an appropriate algorithm for sorting the list by the first two digits (year) of the matriculation number.
SOLUTION: A good solution would be bucket sort. Because the number of digits is known in advance and the number of buckets are at maximum 100. All Students can be sorted within O(n).

Exercise 3 (Hashtables)

a) Give the pseudo code for searching and inserting into a hash table with open addressing. Within the pseudo code $h(k, i)$ represents the hash function and m represents the size of the hash table. Think about how empty can be represented within the hashtable.

SOLUTION:

b) Construct a hash table for the following values by hand: [5, 10, 20002, 40, 2512, 3480, 97, 31]. Use open addressing with linear- and quadratic probing and double hashing. Use $m = 11$ as the size of the hashtable and $h(x) = x \mod 11$ as the hashing function. Assume for quadratic probing $c_1 = 0$ and $c_2 = 1$. For double hashing assume $h2(k) = 1 + k \mod 10$ as the second hash function.

Listing 1 Hash Insert

Input: T: A list that represents the hashtable the new key will be inserted into.
k: the key that should be stored in the hashtable.
Output: i: if it was possible to insert the key the return value will be 1 otherwise 0.

```
begin
  $i := 0$;
  repeat
    $j := h(k, i)$; /* h is the hash function that is used for determining the next place
    for insertion */
    if $T[j] = -1$ then
      $T[j] := k$;
      return 1;
    else
      $i := i + 1$;
  until i=m
  return 0;
end
```

$k = 5$	$[-1, -1, -1, -1, -1, 5, -1, -1, -1, -1, -1]$	insert at 5; 0 collisions
$k = 10$	$[-1, -1, -1, -1, -1, 5, -1, -1, -1, -1, 10]$	insert at 10; 0 collisions
$k = 20002$	$[-1, -1, -1, -1, 20002, 5, -1, -1, -1, -1, 10]$	insert at 4; 0 collisions
$k = 40$	$[-1, -1, -1, -1, 20002, 5, -1, 40, -1, -1, 10]$	insert at 7; 0 collisions
$k = 2512$	$[-1, -1, -1, -1, 20002, 5, 2512, 40, -1, -1, 10]$	insert at 6; 2 collisions
$k = 3480$	$[-1, -1, -1, -1, 20002, 5, 2512, 40, 3480, -1, 10]$	insert at 8; 4 collisions
$k = 97$	$[-1, -1, -1, -1, 20002, 5, 2512, 40, 3480, 97, 10]$	insert at 9; 0 collisions
$k = 31$	$[31, -1, -1, -1, 20002, 5, 2512, 40, 3480, 97, 10]$	insert at 0; 2 collisions

Listing 2 Hash Search

Input: T: A list that represents the hashtable the new key will be inserted into.
k: the key that should be stored in the hashtable.
Output: i: if the value was found the return value will be 1 otherwise 0.

```
begin
  $i := 0$;
  repeat
    $j := h(k, i)$; /* h is the hash function that is used for determining the next place
    for insertion */
    if $T[j] = k$ then
      return 1
    $i := i + 1$;
  until $T[j] = -1 \vee i = m$
  return 0;
end
```

Solution for open addressing with quadratic probing

$k = 5$	$[-1, -1, -1, -1, -1, 5, -1, -1, -1, -1, -1]$	insert at 5; 0 collisions
$k = 10$	$[-1, -1, -1, -1, -1, 5, -1, -1, -1, -1, 10]$	insert at 10; 0 collisions
$k = 20002$	$[-1, -1, -1, -1, 20002, 5, -1, -1, -1, -1, 10]$	insert at 4; 0 collisions
$k = 40$	$[-1, -1, -1, -1, 20002, 5, -1, 40, -1, -1, 10]$	insert at 7; 0 collisions
$k = 2512$	$[-1, -1, -1, -1, 20002, 5, -1, 40, 2512, -1, 10]$	insert at 8; 2 collisions
$k = 3480$	$[-1, -1, 3480, -1, 20002, 5, -1, 40, 2512, -1, 10]$	insert at 2; 3 collisions
$k = 97$	$[-1, -1, 3480, -1, 20002, 5, -1, 40, 2512, 97, 10]$	insert at 9; 0 collisions
$k = 31$	$[-1, -1, 3480, 31, 20002, 5, -1, 40, 2512, 97, 10]$	insert at 3; 4 collisions

Solution for open addressing with double hashing

$k = 5$	$[-1, -1, -1, -1, -1, 5, -1, -1, -1, -1, -1]$	insert at 5; 0 collisions
$k = 10$	$[-1, -1, -1, -1, -1, 5, -1, -1, -1, -1, 10]$	insert at 10; 0 collisions
$k = 20002$	$[-1, -1, -1, -1, 20002, 5, -1, -1, -1, -1, 10]$	insert at 4; 0 collisions
$k = 40$	$[-1, -1, -1, -1, 20002, 5, -1, 40, -1, -1, 10]$	insert at 7; 0 collisions
$k = 2512$	$[-1, -1, 2512, -1, 20002, 5, -1, 40, -1, -1, 10]$	insert at 2; 3 collisions
$k = 3480$	$[-1, -1, 2512, -1, 20002, 5, 3480, 40, -1, -1, 10]$	insert at 6; 2 collisions
$k = 97$	$[-1, -1, 2512, -1, 20002, 5, 3480, 40, -1, 97, 10]$	insert at 9; 0 collisions
$k = 31$	$[31, -1, 2512, -1, 20002, 5, 3480, 40, -1, 97, 10]$	insert at 0; 1 collisions

Exercise 4 (Hashing and C-Compiler)

Assume you want to write a C-compiler. Every C-program consists of identifiers (function-, variable names). As the compiler needs to find identifiers fast, a hash table would be the right choice for storing such identifiers.

a) Estimate an appropriate size for the hash table depending on the size of the input.

SOLUTION:
One could estimate the size of the hashtable as $N \approx \#C/10$, where $\#C$ is the number of characters occurring in the source code.

b) Create a hash function that is able to convert the given identifiers into hash codes.

SOLUTION:
For an identifier $s = s_n \cdots s_0$, one could choose the hash function

$$h(s) = \sum_{i=0}^{n} c(s_i) \cdot p^i \bmod N$$

for a prime number p and N being the size of the hash table, where $c(s_i)$ is the ASCII code associated with a character s_i.

c) Find reasons why the hashing could have a bad runtime behaviour? Give an example program that will have these problems.

SOLUTION:

- There could occur more identifiers than expected in the source code snippet.

  For example, $\#C/10 = 51/10$ underestimates the number of identifiers in the following piece of code:

  ```
  int main(){  int a, b, c, d, e, f, g, h, i, j, k; }
  ```

- If the hash function is known in advance the identifiers could be chosen that each identifier would get into the same place of the hash table.

  For example, in the program

  ```
  #include <stdio.h>

  void exchange(int a[], int i, int j){
   int buf = a[i];
   a[i] = a[j];
   a[j] = buf;
  }

  void bubblesort(int somearray[], int val){
   int i, j;
  ```

```
 for(i = 0; i < val-1; i++){
  for (j=0; j < val-1-i; j++)
   if (somearray[j] > somearray[j+1])
    exchange(somearray, j, j+1);
 }
}

int main() {
 int somearray[] = {1, 4, 5, 7, 9, 2, 34, 1, 78, 9, 0};
 bubblesort(somearray, 11);
 int i;
 for (i=0; i< 11; i++)
  printf("%d ", somearray[i]);
 return 0;
}
```

with a bit more than 400 characters, one could choose $N = 41$. Taking $p = 23$, the identifiers `main` and `val` obtain hash value 22 while `exchange`, `i`, `buf`, `bubblesort` and `somearray` are all mapped to value 23 and `j` gets 24. Especially with linear probing values are significantly clustered.

This can even happen if N is in the magnitude of the number of characters, where the first problem does no longer appear. Consider the following piece of code with 183 characters:

```
int main(){
 int i0, ub, ans = 0;
 printf("Please enter a number: ");
 scanf("%d", &ub); /* read upper bound */
 for (i0 = 0; i0 <= ub; i++)
  ans += i0;
 printf("Sum: %d\n", ans);
}
```

If one chooses $N = 199$ and $p = 1621$, all identifiers (`main`, `i0`, `ans` and `ub`) get hash value 108.