University of Zurich^UZH

Department of Informatics
Database Technology Group
Prof. Dr. M. Böhlen

Informatik II
Assignment 3

Mar 19, 2018

Heap and Heapsort [15 points]

Task 1 [15 points] Create a C program that sorts an array of integers in *ascending* order using heap sort and that prints the sorted array. Your program should first convert the given array into a max-heap and print it in the standard output using the requested format.

Your program should include the following:

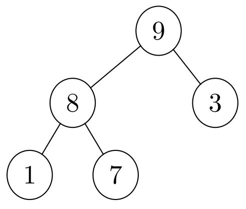
1. The function *void buildMaxHeap(int A[], int n)*, where *A* is the array to be converted, and *n* is the real size of the array.
2. The function *void printHeap(int A[], int n)* that prints the created max-heap to the console in the format `graph g {` (all the edges in the form `NodeA -- NodeB`) `}`, where each edge should be printed in a separate line. The ordering of the edges is not relevant.
3. The function *void heapSort(int A[], int n)* that sorts the array *A* in ascending order.
4. The function *void printArray(int A[], int n)* that prints a given array to the console.

For example, given the array `A = [3, 8, 9, 1, 7]` your program should produce the max-heap `[9, 8, 3, 1, 7]` and print it to the console in the form illustrated on the left figure. Print the content of array *A* after calling *heapSort* on it. The alternative output is used to check the correctness of your implementation, it does not have to be part of your solution.

Output Form

```
graph g {
  9 -- 8
  9 -- 3
  8 -- 1
  8 -- 7
}
```

Alternative Output Form



Test your program with the array `[4, 3, 2, 5, 6, 7, 8, 9, 12, 1]`.

<u>**Note:**</u> You can copy the output of your program and use it on `http://www.webgraphviz.com/` to view your max-heap in the form of a binary tree, as illustrated on the right figure.

Quicksort [20 points]

Task 2. [20 points]

(a) [12 points] Create a C program that sorts an array of integers in *ascending* order using quicksort and that prints the sorted array. Your program should include the following functions:

  i. *void swap(int A[], int i, int j)* that exchanges the element with index i with the element with index j in array A.

  ii. *int partitionHoare(int A[], int low, int high)* that rearranges the area of array A restricted by indices $low, high$ based on the element $A[high]$, that is chosen as the pivot.

  iii. *void quicksort(int A[], int low, int high)* that sorts the area of array A restricted by indices $low, high$.

  iv. *void printArray(int A[], int size)* that prints the array A containing $size$ elements.

  Test your program with the array `[4, 3, 2, 5, 6, 7, 8, 9, 12, 1]`.

(b) [8 points] The efficiency of quicksort is influenced by the choice of the pivot. In this task, you are asked to offer the possibility to switch between two different ways of choosing the pivot: as the last element of the given array (as was already done in the previous task) or as the median of the first, the last and the middle element. Extend the program you created in Task 2 so that function `quicksort` has one more argument `int choice` that determines the way the pivot will be chosen and can be assigned the values 1 or 0. If `choice=1`, the function shall check the three array elements $A[low]$, $A[\lfloor \frac{low+high}{2} \rfloor]$ and $A[high]$ and use their median as a pivot.

  For example, the call `quicksort(A, 1, 6, 1)` for the array `A = [8, 4, 2, 0, 9, 6, 3, 7]` would use as a pivot the value 3 since this is the median of $A[1] = 4$, $A[\lfloor \frac{1+6}{2} \rfloor] = A[3] = 0$ and $A[6] = 3$.

Comparing sorting algorithms [10 points]

Task 3. [10 points] You are given 3 integer datasets in files "ordered.txt", "inverse.txt", "random.txt". Each file contains 30000 integers but in different order. "ordered.txt" and "inverse.txt" contain integers in ascending and descending order accordingly. "random.txt" contains the same numbers in random order. This task is about comparing the efficiency of different sorting algorithms for these three cases. Run your heap-sort and quicksort implementations and bubble sort (you can use your implementation from Task 4 of Exercise 1) on these datasets and report their runtime as following.

	ordered	random	inverse
heap sort			
quick sort (simple)			
quick sort (median of 3)			
bubble sort			

Note 1: In order to read the given files use the following function *void readFile(char filename[], int output[], int *n)*. *filename* is the name of an input file, *output* is the output array containing integers of the input file, *n* stores the size of the array (number of elements in the input file).

```c
void readFile(char filename[], int output[], int *n) {
    FILE *f;
    int i;

    f=fopen(filename, "r");

    i=0;
    while(fscanf(f, "%d", &output[i]) == 1) i++;
    *n=i-1;

    fclose(f);
}
```

Note 2: Use the following part of code to measure elapsed time for sorting algorithms:

```c
#include <time.h>

clock_t start;
clock_t end;
float seconds;

start = clock();
// your sorting function
end = clock();

seconds = (float)(end - start) / CLOCKS_PER_SEC;
printf("secs: %f\n", seconds);
```

Linked Lists [15 points]

Task 4. [15 points] Write a C program that implements singly-linked lists of integers defined as follows:

```c
struct list {
  struct node* head;
};

```

```c
struct node {
  int val;
  struct node* next;
};

```

Your program must contain the following functions:

a) *struct list* init()* that initializes a list.

b) *void append(struct list *listA, int val)* that appends a new node containing integer `val` at the end of the list.

c) *void reverse(struct list *listA)* that reverses the list in O(n) time.

d) *void clear(struct list *listA)* that removes all nodes from the list.

e) *void print(struct list *listA)* that prints the values of the elements of the list to the console enclosed in brackets, e.g. `[ 3 5 7 2 ]`.

f) *void delete(struct list *listA, int i)* that removes the node in the *i-th* position in the list. The index of the first element is 0.

g) *void deleteLast(struct list *listA)* that removes the last node in the list.

h) *void max(struct list *listA)* that finds the element with the maximum key value and prints the key in the console after the message "`max =`".

IMPORTANT: Every time you add or remove a node make sure you request (malloc) and free (free) the memory involved, correspondingly.

After you have implemented and tested these functions with lists of your choice, include the following sequence in your *main()* method:

1. Insert 9,4,5,3,1,2,0 to the list, in the given order.
2. Print the list to the console.
3. Reverse the list.
4. Print the list to the console.
5. Remove the nodes in positions 6, 3 and 0 from the list, one after another.
6. Print the list to the console.
7. Remove the last element from the list.
8. Print the list to the console.
9. Find the node with the maximum value and print the value to the console.
10. Clear the list.
11. Print the list to the console.

Submission

For this exercise, you need to submit a zipped folder *a<exercise number>_<family name>_<matriculation number>.zip* where `family name` and `matriculation number` correspond to your personal data. This folder should include:

a) the C-files you created for each of the tasks. Each C-file should be named as *task<task number>.c*

b) a pdf named *a<exercise number>.pdf* with the solutions for Task 3.

Make sure that both in the C-files as well as in the pdf file you submit, your personal data is included (in the form of comments or a note).

Deadline: **Sunday, April 1$^{\text{st}}$ at 23:59**.