**Output:**

Sorted by Name:

Name: Eve, Age: 22, Height: 5.70

Name: Dave, Age: 20, Height: 5.90

Name: Charlie, Age: 35, Height: 5.60

Name: Bob, Age: 25, Height: 5.80

Name: Alice, Age: 30, Height: 5.50

Sorted by Age:

Name: Charlie, Age: 35, Height: 5.60

Name: Alice, Age: 30, Height: 5.50

Name: Bob, Age: 25, Height: 5.80

Name: Eve, Age: 22, Height: 5.70

Name: Dave, Age: 20, Height: 5.90

Sorted by HeightName: Dave, Age: 20, Height: 5.90

Name: Bob, Age: 25, Height: 5.80

Name: Eve, Age: 22, Height: 5.70

Name: Charlie, Age: 35, Height: 5.60

Name: Alice, Age: 30, Height: 5.50

**HW07.c**

#include <stdio.h>

#include "heap.h"

#include <string.h>

typedef struct Person\_ {

    const char \*name;

    int age;

    double height;

} Person;

void outputSorted(const Person people[], int numPeople, int (\*compare)(const void \*pKey1, const void \*pKey2)) {

    Heap heap;

    heap\_init(&heap, compare, NULL);

    // Insert people into the heap

    for (int i = 0; i < numPeople; i++) {

        heap\_insert(&heap, &people[i]);

    }

    // Extract and output each person in sorted order

    void \*data;

    while (heap\_size(&heap) > 0) {

        heap\_extract(&heap, &data);

        Person \*person = (Person \*)data;

        printf("Name: %s, Age: %d, Height: %.2f\n", person->name, person->age, person->height);

    }

    heap\_destroy(&heap);

}

int compareByName(const void \*pKey1, const void \*pKey2) {

    Person \*person1 = (Person \*)pKey1;

    Person \*person2 = (Person \*)pKey2;

    return strcmp(person1->name, person2->name);

}

int compareByAge(const void \*pKey1, const void \*pKey2) {

    Person \*person1 = (Person \*)pKey1;

    Person \*person2 = (Person \*)pKey2;

    return person1->age - person2->age;

}

int compareByHeight(const void \*pKey1, const void \*pKey2) {

    Person \*person1 = (Person \*)pKey1;

    Person \*person2 = (Person \*)pKey2;

    if (person1->height < person2->height) return -1;

    if (person1->height > person2->height) return 1;

    return 0;

}

int main(){

    Person people[] = {

    {"Alice", 30, 5.5},

    {"Bob", 25, 5.8},

    {"Charlie", 35, 5.6},

    {"Dave", 20, 5.9},

    {"Eve", 22, 5.7}};

printf("Sorted by Name:\n");

outputSorted(people, 5, compareByName);

printf("\nSorted by Age:\n");

outputSorted(people, 5, compareByAge);

printf("\nSorted by Height");

outputSorted(people, 5, compareByHeight);

return 0;

}

**Heap.h**

/\*

 \* heap.h

 \*/

#ifndef HEAP\_H

#define HEAP\_H

/\* Define a structure for heaps. \*/

typedef struct Heap\_ {

    int size;

    int (\*compare)(const void \*key1, const void \*key2);

    void (\*destroy)(void \*data);

    void \*\*tree;

} Heap;

/\* Public Interface \*/

void heap\_init(Heap \*heap, int(\*compare)(const void \*key1, const void \*key2),

        void(\*destroy)(void \*data));

void heap\_destroy(Heap \*heap);

int heap\_insert(Heap \*heap, const void \*data);

int heap\_extract(Heap \*heap, void \*\*data);

#define heap\_size(heap) ((heap)->size)

#endif

**Heap.c**

/\*

 \* heap.c

 \*/

#include <stdlib.h>

#include <string.h>

#include "heap.h"

/\* Define private macros used by the heap implementation. \*/

#define heap\_parent(npos) ((int)(((npos) - 1) / 2))

#define heap\_left(npos) (((npos) \* 2) + 1)

#define heap\_right(npos) (((npos) \* 2) + 2)

void heap\_init(Heap \*heap, int(\*compare)(const void \*key1, const void \*key2),

        void(\*destroy)(void \*data)) {

    /\* Initialize the heap. \*/

    heap->size = 0;

    heap->compare = compare;

    heap->destroy = destroy;

    heap->tree = NULL;

}

void heap\_destroy(Heap \*heap) {

    int i;

    /\* Remove all the nodes from the heap. \*/

    if (heap->destroy != NULL) {

        for (i = 0; i < heap\_size(heap); i++) {

            /\* Call a user-defined function to free dynamically allocated

             \* data. \*/

            heap->destroy(heap->tree[i]);

        }

    }

    /\* Free the storage allocated for the heap. \*/

    free(heap->tree);

    /\* No operations are allowed now, but clear the structure as a

     \* precaution. \*/

    memset(heap, 0, sizeof(Heap));

}

int heap\_insert(Heap \*heap, const void \*data) {

    void \*temp;

    int ipos, ppos;

    /\* Allocate storage for the node. \*/

    if ((temp = (void \*\*) realloc(heap->tree, (heap\_size(heap) + 1) \* sizeof(void \*))) == NULL) {

    return -1;

} else {

    heap->tree = (void \*\*) temp;

}

    /\* Insert the node after the last node. \*/

    heap->tree[heap\_size(heap)] = (void \*) data;

    /\* Heapify the tree by pushing the contents of the new node upward. \*/

    ipos = heap\_size(heap);

    ppos = heap\_parent(ipos);

    while (ipos > 0 && heap->compare(heap->tree[ppos], heap->tree[ipos]) < 0) {

        /\* Swap the contents of the current node and its parent. \*/

        temp = heap->tree[ppos];

        heap->tree[ppos] = heap->tree[ipos];

        heap->tree[ipos] = temp;

        /\* Move up one level in the tree to continue heapifying. \*/

        ipos = ppos;

        ppos = heap\_parent(ipos);

    }

    /\* Adjust the size of the heap to account for the inserted node. \*/

    heap->size++;

    return 0;

}

int heap\_extract(Heap \*heap, void \*\*data) {

    void \*save, \*temp;

    int ipos, lpos, rpos, mpos;

    /\* Do not allow extraction from an empty heap. \*/

    if (heap\_size(heap) == 0)

        return -1;

    /\* Extract the node at the top of the heap. \*/

    \*data = heap->tree[0];

    /\* Adjust the storage used by the heap. \*/

    save = heap->tree[heap\_size(heap) - 1];

    if (heap\_size(heap) - 1 > 0) {

        if ((temp = (void \*\*) realloc(heap->tree, (heap\_size(heap) - 1) \* sizeof(void \*))) == NULL) {

    return -1;

} else {

    heap->tree = (void \*\*) temp;

}

        /\* Adjust the size of the heap to account for the extracted node. \*/

        heap->size--;

    } else {

        /\* Manage the heap when extracting the last node. \*/

        free(heap->tree);

        heap->tree = NULL;

        heap->size = 0;

        return 0;

    }

    /\* Copy the last node to the top. \*/

    heap->tree[0] = save;

    /\* Heapify the tree by pushing the contents of the new top downward. \*/

    ipos = 0;

    lpos = heap\_left(ipos);

    rpos = heap\_right(ipos);

    while (1) {

        /\* Select the child to swap with the current node. \*/

        lpos = heap\_left(ipos);

        rpos = heap\_right(ipos);

        if (lpos < heap\_size(heap) && heap->compare(heap->tree[lpos],

                heap-> tree[ipos]) > 0) {

            mpos = lpos;

        } else {

            mpos = ipos;

        }

        if (rpos < heap\_size(heap) && heap->compare(heap->tree[rpos],

                heap-> tree[mpos]) > 0) {

            mpos = rpos;

        }

        /\* When mpos is ipos, the heap property has been restored. \*/

        if (mpos == ipos) {

            break;

        } else {

            /\* Swap the contents of the current node and the selected child. \*/

            temp = heap->tree[mpos];

            heap->tree[mpos] = heap->tree[ipos];

            heap->tree[ipos] = temp;

            /\* Move down one level in the tree to continue heapifying. \*/

            ipos = mpos;

        }

    }

    return 0;

}