Data Structures

Binary Heap Implementation in C

Dimitrios Michail

Dept. of Informatics and Telematics
Harokopio University of Athens

Min Heap

as an abstract-data-type

A minimum heap is an abstract data type which includes the following operations:

- Insert a new element x with key k, INSERT(H,x,k).
- Find the element with the smallest key (highest priority), FINDMIN(H).
- Delete the element with the smallest key (highest priority), DELMIN(H).
- Return the number of elements in the heap, SIZE(H)
- Check if the heap is empty, ISEMPTY(H).

Binary Heap

heap-ordered

A tree is heap-ordered if for any node v the key of v is smaller or equal to the key of its children.

Binary Heap

heap-ordered

A tree is heap-ordered if for any node v the key of v is smaller or equal to the key of its children.

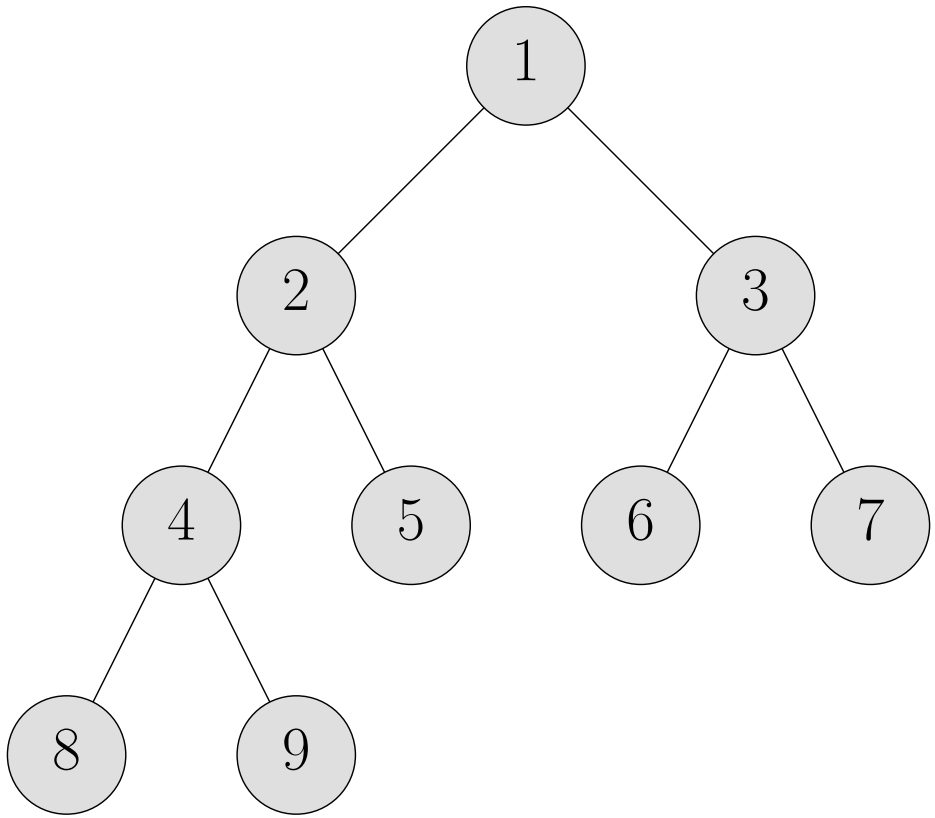
Binary Heap

A **binary heap** is a set of nodes with keys placed on a complete binary tree which is heap-ordered and represented as an array.

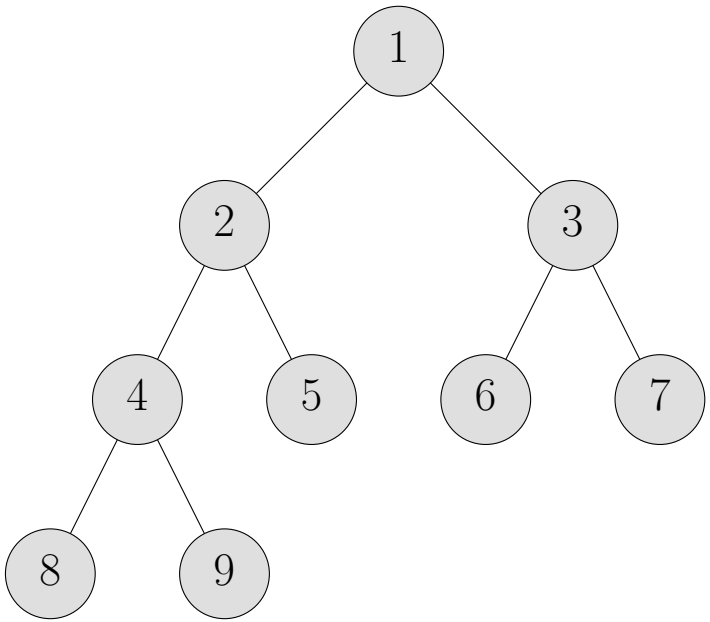
Complete Binary Tree

Definition

A binary tree where all levels, except maybe the last, are full. The last level of the tree if not complete, is filled from left to right.



Complete Binary Tree as an Array



1	2	3	4	5	6	7	8	9

- $parent(i) = \lfloor i/2 \rfloor$
- $left - child(i) = 2i$
- $right - child(i) = 2i + 1$

Binary Heap

interface in C (minheap.h)

```
#ifndef _MINHEAP_H
#define _MINHEAP_H

typedef int key_type;
typedef struct _minheap* minheap;

minheap minheap_create();
minheap minheap_heapify(const key_type* array, int n);
void    minheap_destroy(minheap);

int     minheap_findmin(minheap);
void    minheap_insert(minheap, key_type);
void    minheap_deletemin(minheap);

int     minheap_is_empty(minheap);
int     minheap_size(minheap);
void    minheap_clear(minheap);

#endif
```

Binary Heap Implementation

Representation (minheap.c)

```
#include <stdio.h>
#include <stdlib.h>
#include <assert.h>
#include "minheap.h"

struct _minheap {
    key_type* array;
    int max_size;
    int cur_size;
};
```

1. `array` is the array for the keys
2. `max_size+1` is the array size
3. `cur_size` is the position of the last array element which is used

Binary Heap Implementation

Create (minheap.c)

```
minheap minheap_create() {
    minheap h = (minheap) malloc(sizeof(struct _minheap));
    if (h == NULL) {
        fprintf(stderr, "Not enough memory!\n");
        abort();
    }

    h->max_size = 64;
    h->cur_size = 0;
    h->array = (key_type*) malloc( \
        sizeof(key_type)*(h->max_size+1));
    if (h->array == NULL) {
        fprintf(stderr, "Not enough memory!\n");
        abort();
    }

    return h;
}
```

Binary Heap Implementation

Destruction (minheap.c)

```
void minheap_destroy(minheap h) {
    assert(h);
    free(h->array);
    free(h);
}
```

Binary Heap Implementation

Double Capacity (minheap.c)

```
static void minheap_double_capacity(minheap h) {

    // create double the array
    int new_max_size = 2 * h->max_size;
    key_type* new_array = (key_type*) malloc( \
        sizeof(key_type)*(new_max_size+1));
    if (new_array == NULL) {
        fprintf(stderr, "Not enough memory!\n");
        abort();
    }

    /* copy old elements to new array */
    for(int i = 1; i <= h->cur_size; i++) {
        new_array[i] = h->array[i];
    }

    /* free old array and place new in position */
    free(h->array);
    h->array = new_array;
    h->max_size = new_max_size;
}
```

Binary Heap Implementation

Swap Elements (minheap.c)

```
static
void minheap_swap(minheap h, int i, int j) {
    assert (h && i >=1 && i <= h->cur_size &&
            j >= 1 && j <= h->cur_size);
    key_type tmp = h->array[i];
    h->array[i] = h->array[j];
    h->array[j] = tmp;
}
```

Binary Heap Implementation

Fixup (minheap.c)

```
static
void minheap_fixup(minheap h, int k) {
    assert(h && k >= 1 && k <= h->cur_size);

    while (k>1 && h->array[k] < h->array[k/2]) {
        minheap_swap(h, k/2, k);
        k /= 2;
    }
}
```

Binary Heap Implementation

Fixdown (minheap.c)

```
static
void minheap_fixdown(minheap h, int k) {
    assert(h);

    while (2*k <= h->cur_size) {
        int j = 2*k;
        if (j < h->cur_size && h->array[j+1] < h->array[j])
            j++;
        if (h->array[k] <= h->array[j])
            break;

        minheap_swap(h, k, j);
        k = j;
    }
}
```

Binary Heap Implementation

Insert (minheap.c)

```
void minheap_insert(minheap h, key_type key) {
    assert(h);

    // make sure there is space
    if (h->cur_size == h->max_size)
        minheap_double_capacity(h);

    // add at the bottom, as a leaf
    h->array[++h->cur_size] = key;

    // fix its position
    minheap_fixup(h, h->cur_size);
}
```

Binary Heap Implementation

Find Minimum (minheap.c)

```
int minheap_findmin(minheap h) {
    if (minheap_is_empty(h)) {
        fprintf(stderr, "Heap is empty!\n");
        abort();
    }

    // min is always in first position
    return h->array[1];
}
```

Binary Heap Implementation

Delete Minimum (minheap.c)

```
void minheap_deletemin(minheap h) {
    if (minheap_is_empty(h)) {
        fprintf(stderr, "Heap is empty!\n");
        abort();
    }

    // swap first with last
    minheap_swap(h, 1, h->cur_size);

    // delete last
    h->cur_size--;

    // fixdown first
    minheap_fixdown(h, 1);
}
```

Binary Heap Implementation

Size and Is-Empty (minheap.c)

```
int minheap_size(minheap h) {
    assert(h);
    return h->cur_size;
}

int minheap_is_empty(minheap h) {
    assert(h);
    return h->cur_size <= 0;
}
```

Binary Heap Implementation

Clear (minheap.c)

```
void minheap_clear(minheap h) {
    assert(h);
    h->cur_size = 0;
}
```

Binary Heap Implementation

Heapify (minheap.c)

```
minheap minheap_heapify(const key_type* array, int n) {
    assert(array && n > 0);

    minheap h = (minheap) malloc(sizeof(struct _minheap));
    if (h == NULL) {
        fprintf(stderr, "Not enough memory!\n");
        abort();
    }
    h->max_size = n;
    h->cur_size = 0;
    h->array = (key_type*) malloc(sizeof(key_type)*(h->max_size+1));
    if (h->array == NULL) {
        fprintf(stderr, "Not enough memory!\n");
        abort();
    }

    h->cur_size = n;
    for(int k = 0; k < n; k++)
        h->array[k+1] = array[k];

    for(int k = (h->max_size+1)/2; k > 0; k--)
        minheap_fixdown(h, k);

    return h;
}
```

Using the Heap

```c
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
#include "minheap.h"

int main() {
    int i;
    srand(time(NULL));

    minheap h = minheap_create();

    for(i = 0; i < 100; i++)
        minheap_insert(h, rand() % 1000);

    while(!minheap_is_empty(h)) {
        printf("%4d", minheap_findmin(h));
        minheap_deletemin(h);
    }

    minheap_destroy(h);

    return 0;
}
```

HeapSort

```c
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
#include "minheap.h"

void heapsort(int *array, int n) {
    minheap h = minheap_heapify(array, n);

    int i = 0;
    while(!minheap_is_empty(h)) {
        array[i++] = minheap_findmin(h);
        minheap_deletemin(h);
    }

    minheap_destroy(h);
}
```

HeapSort (continued)

```
int main() {
    srand(time(NULL));

    int array[SIZE];
    for(int i = 0; i < SIZE; i++) {
        array[i] = rand() % MAX_NUMBER;
    }

    heapsort(array, SIZE);

    for(int i = 1; i < SIZE; i++) {
        assert(array[i-1] <= array[i]);
    }
    return 0;
}
```