Structuri de Date
Laboratorul 8: Heaps

Dan Novischi

15 aprilie 2019

1. Introducere

Scopul acestui laborator îl reprezintă implementarea interfeței de lucru cu o coada (maximă) de piorități bazată pe structura de Heap studiată în cadrul cursului.

În prezentul laborator, structura (binară) de Heap este un array de elemente care poate fi vizualizat conceptual ca un arbore binar complet (vezi Figura 1), unde:

- fiecare nod din arbore corespunde unui element stocat în array
- relațiile de ordine între nodurile arborelui sunt date de:

```
/* Max Heap */
Element[parinte] > Element[copil[stanga]]
Element[parinte] > Element[copil[dreapta]]

/* Min Heap */
Element[parinte] < Element[copil[stanga]]
Element[parinte] < Element[copil[dreapta]]
```

- relativ la indexul din array, începand numerotarea de la zero, pentru parinți si copii avem relelațiile:

$$Parent(i) = \lfloor (i-1)/2 \rfloor$$

$$Left(i) = 2i + 1$$

$$Right(i) = 2i + 2$$

- proprientătile array-ului reprezentand Heap-ul sunt: **numarul de elemente** stocate curent în array și dimensiunea totală a array-ului.

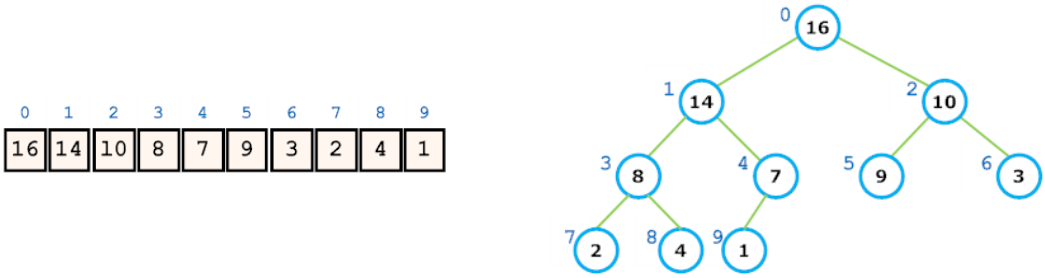

Figura 1: Max Heap: *array - dreapta, respectiv arbore - stanga*

Astfel, pentru reprezentarea unei cozi maxime de priorități avem urmatoarele definții:

```
typedef struct {
  int prior; // element priority
  Item data; // element data
}ItemType;
```

```
typedef struct heap{
  long maxHeapSize; // array size
  long size; // number of elements
  ItemType *elem; // array of elements
} PriQueue, *APriQueue;
```

2. Cerințe

Implementați umătoarele funcții:

- `makeQueue` - creeaza un Heap nou.
- `getLeftChild` - returnează indexul copilului din stanga.
- `getRightChild` - returnează indexul copilului din dreapta.
- `getParent` - returneză indexul parintelui.
- `sifttUp` - reface/păstrează proprietatea de Heap parcurgand pe "link-uri" de parinte.
- `insert` - insereaza un nou element în Heap menținand proprietățile acestuia.
- `getMax` - returnează elementul de prioritate maximă din Heap.
- `siftDown` - reface/păstrează proprietatea de Heap parcurgand pe link-uri de copii.
- `removeMax` - șterge elementul de prioritate maximă din Heap menținand proprietățile acestuia.
- `freeQueue` - distruge Heap-ul eliberand memoria utilizată.

Observație: Pentru a verifca corectitudinea implementării utilizați `make test`. Output unei implementări corecte arată ca mai jos.

```
...$ make test
valgrind --leak-check=full ./TestHeap
==6512== Memcheck, a memory error detector
==6512== Copyright (C) 2002-2015, and GNU GPL'd, by Julian Seward et al.
==6512== Using Valgrind-3.11.0 and LibVEX; rerun with -h for copyright info
==6512== Command: ./TestHeap
==6512==
. Testul MakeQueue a fost trecut cu succes!      Puncte: 0.10
. Testul GetParent a fost trecut cu succes!      Puncte: 0.05
. Testul GetLeftChild a fost trecut cu succes!   Puncte: 0.05
. Testul getRightChild a fost trecut cu succes!  Puncte: 0.05
. Testul testSiftUp a fost trecut cu succes!     Puncte: 0.20
. Testul testInsert a fost trecut cu succes!     Puncte: 0.10
. Testul testGetMax a fost trecut cu succes!     Puncte: 0.05
. Testul testSiftDown a fost trecut cu succes!   Puncte: 0.20
. Testul testRemoveMax a fost trecut cu succes!  Puncte: 0.10
. Testul FreeQueue: *Se va verifica cu valgrind* Puncte: 0.10.

Scor total: 1.00 / 1.00

==6512==
==6512== HEAP SUMMARY:
==6512==     in use at exit: 0 bytes in 0 blocks
==6512==   total heap usage: 4 allocs, 4 frees, 1,096 bytes allocated
==6512==
==6512== All heap blocks were freed -- no leaks are possible
==6512==
==6512== For counts of detected and suppressed errors, rerun with: -v
==6512== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from 0)
```