Structuri de date

Rotaru Alexandru Andrei
rotarualexandruandrei94@gmail.com
2019-2020

University Politehnica of Bucharest

1 Arbori

1.1 Cozi de priorități

În multe situații avem nevoie sa putem accesa cel mai mic/mare element dintr-o mulțime. O structura generala care oferă astfel de operație se numește coada de priorități (priority queue). Principalele operații asociate cu o astfel de structura sunt:

- push inserează un element
- pop elimina elementul cel mai mic/mare
- top returnează elementul cel mai mic/mare

Scenarii de utilizare:

- algoritmul lui Dijkstra de gasire a celei mai scurte cai intr-un graf
- algoritmi de compresie (codificare Huffman)
- inteligenta artificiala (Algoritmul de cautare A*)
- sisteme de operare: coada de procese ce trebuie planificate pe procesor
- sisteme distribuite: load balancing

1.2 Heap

Heap-ul este cea mai populara implementare de cozi de priorități și oferă operații la un nivel de complexitate comparabil cu a unui arbore binar de căutare însa funcția de găsire a minimului/maximului se face în timp constant. Heap-ul este un arbore (de obicei binar) complet reprezentat sub forma vectoriala. Legăturile intre părinti și copii sunt deduse pe baza formulelor 1, 2, 3.

$$left\_child\_index = index * 2 + 1 \tag{1}$$

$$right\_child\_index = index * 2 + 2 \tag{2}$$

$$parent\_index = \frac{index - 1}{2} \tag{3}$$

Una din proprietățile importante ale heap-urilor în funcție de tipul lor (min-heap/max-heap) este faptul ca valorile fiilor sunt mai mici sau egale decât a părintelui într-un max-heap / mai mari într-un min-heap.

În figura 1 puteți observa cum un arbore binar poate fi reprezentat pe un vector. Un alt efect important al acestui mod de reprezentare este faptul ca heap-urile sunt arbori binar balansați. Indiferent de operațiile pe care le facem proprietatea de arbore balansat se păstrează deoarece la orice moment de timp dat diferența maxima de înălțimi de frunze este 1.

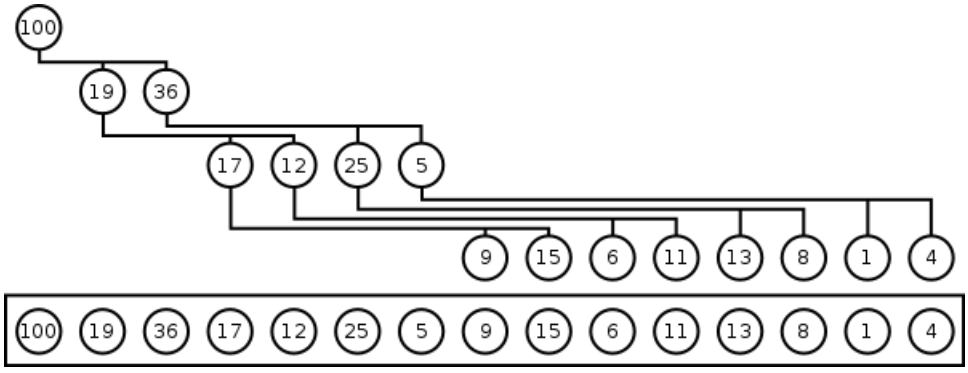

Figure 1: Max-heap views: array vs binary tree

2 Precizări

Laborantul va preciza ce exerciții veți avea de rezolvat. Pentru rezolvarea fiecărui exercițiu veți folosi un fișier separat. Fișierele sunt deja create și conțin declarațiile de funcții.

Listing 1: Rularea întregului set de exerciții

```
1 make all
```

Listing 2: Rularea unui subset de exerciții

```
1 make build
2 ./lab 1 2 3 4
```

3 Resurse

Laborantul va preciza ce exercitii veti avea de rezolvat. Pentru rezolvarea fiecarui exercitiu veti folosi un fisier separat. **Optional**, va puteti verifica folosind checker-ul (daca este pus la dispozitie).

3.1 Vizualizare

Pentru a putea visualiza o parte din algoritmi si intelege mai bine cum functioneaza structurile de date accesati:
`https://www.cs.usfca.edu/~galles/visualization/Algorithms.html`

3.2 Link-uri catre platforme online cu probleme

- Hackerrank: `https://www.hackerrank.com/domains/data-structures`
- Leetcode: `https://leetcode.com/problemset/all/`

3.3 Tool-uri de debug

- GDB: `https://cs.baylor.edu/~donahoo/tools/gdb/tutorial.html`
- Valgrind: `http://valgrind.org/docs/manual/quick-start.html`

3.4 Feedback

Pentru a semnala probleme sau a oferi sugestii pentru laboratoarele urmatoare:
`https://forms.gle/8bRw7mPgqaJ9tRzd6`

4 Exerciții

4.1 Exercițiul 1 [2 puncte]

Un caz de folosirea heap-urilor este heap-sort. Deoarece top returnează elementul cel mai mic/mare putem sa obținem elementele în ordine scoțându-le pe rand din heap.

Pentru a acest exercițiu vom folosi o implementare standard de cozi de priorități din standard template library (stl). Deoarece stl este disponibil doar în C++ sintaxa de folosire se schimba puțin.

Listing 3: priority_queue

```
1  /* declararea unei cozi de prioritati de tip intreg*/
2  std::priority_queue<int> coada_de_prioritati;
3
4  /*inserarera unui element in coada de prioritati*/
5  coada_de_prioritati.push(2);
6
7  /*accesul elementului cel mai mare,
8  in C++ coada de prioritati este max-heap*/
9  int biggest = coada_de_prioritati.top();
10
11 /*eliminarea din coada a celui mai mare element*/
12 coada_de_prioritati.pop();
```

Funcția va primi ca parametru un vector de întregi și o lungime și va modifica elementele astfel încât sa devina ordonate crescător.

Listing 4: Semnăturile funcțiilor

```
1  void sort_using_pq_heaps(int *values, size_t length);
```

4.2 Exercițiul 2 [10 puncte]

Fiind data structura de heap sa se implementeze operațiile

- push: inserează un element în heap
- top: returnează elementul cel mai mic din heap
- pop: elimina elementul cel mai mic din heap

Listing 5: Semnăturile funcțiilor

```
typedef struct binary_heap {
    int *values;
    size_t capacity;
    size_t length;
} binary_heap_t;

void heap_push(binary_heap_t * const heap,
    const int value);
int heap_top(binary_heap_t const * const heap);
void heap_pop(binary_heap_t * const heap);
```

Listing 6: Pseudocod push

```
percolate_up(heap, current) {
    while (current element < parent element)
        swap current with parent
        index becomes parent index
}

heap_push(heap, value) {
    ensure_space(heap);
    heap[size++] = value;
    percolate_up(heap, size - 1);
}
```

Listing 7: Pseudocod top

```
top(heap) {
    return heap[0];
}
```

Listing 8: Pseudocod pop

```
percolate_down(heap, current) {
    minimum_child_idx = current

    left_child = 2 * current + 1
    right_child = 2 * current + 2

    if left_child < size
        and heap[minimum_child_idx] < heap[left_child]
        minimum_child_idx = left_child

    if right_child < size
        and heap[minimum_child_idx] < heap[right_child]
        minimum_child_idx = right_child

    swap current with minimum_child_idx
    percolate_down(heap, minimum_child_idx)
}

heap_push(heap, value) {
    swap heap[0], heap[size - 1];
    size -= 1;
    percolate_down(heap, 0);
}
```

4.3 Exercițiul 3 [3 puncte]

Sa se implementeze funcția de ștergere a unui element dat dintr-un heap. În cazul în care elementul nu este prezent structura rămâne neschimbata.

Listing 9: Semnături de funcții

```
void heap_remove(binary_heap_t * const heap,
    const int value);
```

5 Extra/Bonus

Aceste exerciții trebuie implementate împreună cu o suita de teste pentru a putea fi validate. Puteți prezenta ideea voastră asistentului și sa primiți puncte în funcție de creativitate.

5.1 Exercițiul 4 [5 puncte]

Se considera un vector 'aproape' sortat cu proprietatea ca orice element se afla la cel mult k poziții de poziția finala dacă vectorul ar fi sortat. Propuneți un algoritm cat mai eficient de sortare a vectorului.

5.2 Exercițiul 5 [5 puncte]

Propuneți un algoritm eficient de a transforma un max-heap în min-heap.