2-INF-237 Vybrané partie z dátových štruktúr

2-INF-237 Selected Topics in Data Structures

- Instructor: Broňa Brejová
- E-mail: brejova@fmph.uniba.sk
- Office: M163

- Course webpage: http://compbio.fmph.uniba.sk/vyuka/vpds/

Priority queues

Min-heap property

A tree or a forest has the min-heap property, if the key in each non-root node $\geq$ key in its parent

Running time of heap operations

Operations	Bin. heap worst case	Fib. heap amortized
MakeHeap()	$O(1)$	$O(1)$
Insert(H, x)	$O(\log n)$	$O(1)$
Minimum(H)	$O(1)$	$O(1)$
ExtractMin(H)	$O(\log n)$	$O(\log n)$
Union(H_1, H_2)	$O(n)$	$O(1)$
DecreaseKey(H, x, k)	$O(\log n)$	$O(1)$
Delete(H, x)	$O(\log n)$	$O(\log n)$
BuildHeap($x_1, \ldots, x_n$)	$O(n)$	$O(n)$

Note: DecreaseKey and Delete require a handle of the element (pointer to node/index to array)

Applications of priority queues

- Job scheduling on a server (Insert, ExtractMin, Delete)
- Heapsort (BuildHeap, ExtractMin)
- Bentley–Ottmann algorithm for finding intersections of line segments (Insert, ExtractMin)
- Kruskal's and Prim's algorithms for minimum spanning tree (BuildHeap, ExtractMin, DecreaseKey)
- Dijkstra's algorithm for shortest paths (BuildHeap, ExtractMin, DecreaseKey)

Queues and Sorting

Heapsort: BuildHeap or $n \times$ Insert, then $n \times$ ExtractMin

Lower bound: on Insert+ExtractMin $\Omega(\log n)$ in the comparison model

Partial sorting: given n elements, print smallest k in the sorted order

$O(n + k \log k)$ - how?

Incremental sorting: given n elements, build a data structure, then support ExtractMin called k times, k unknown in advance

– with heaps $O(n + k \log n)$

– this is actualy $O(n + k \log k)$: check for $k \leq \sqrt{n}$ and $k \geq \sqrt{n}$

Bentley–Ottmann algorithm
for finding intersections of line segments

n line segments, k intersections

Sweepline algorithm, sweeps from left to right

Maintains priority queue of events: start of a line, end of a line, intersection

Balanced binary search tree of segments at current x-coordinate

$(2n + k)\times$Insert, $(2n + k)\times$ExtractMin

$O((n + k) \log n)$ time

Kruskal's algorithm for minimum spanning trees

Process edges from the smallest weight upwards

If edge connects 2 components, add it

Finish when we have a tree.

Instead of sorting edges:

BuildHeap, at most $m\times$ ExtractMin

Also needs Union-FindSet structure

Total time $O(m \log n)$

Prim's algorithm for minimum spanning trees

Build a tree starting from one node

For each node outside the tree keep the best cost to a node in the tree as a priority in a heap

BuildHeap, $n\times$ Extract-Min, $m\times$ DecreaseKey

$O(m \log n)$ with binary heaps, $O(m + n \log n)$ with Fibonacci heaps

Note: minimum spanning tree can be found in $O(m\alpha(m, n))$

where α is the inverse of Ackermann's function

Using Soft heaps by Chazelle 2000

Constant amortized time for insert, union, findMin, delete

But corrupts (increases) keys of up to ϵn elements

Dijkstra's algorithm for shortest paths

Heap of nodes without final distance

In each step step choose the minimum, set its distance to final, update distances of its neighbors

BuildHeap, $n\times$ Extract-Min, $m\times$ DecreaseKey

$O(m \log n)$ with binary heaps, $O(m + n \log n)$ with Fibonacci heaps

Fibonacci heaps

Michael L. Fredman, Robert E. Tarjan. "Fibonacci heaps and their uses in improved network optimization algorithms."Journal of the ACM 1987

Introduction to Algorithms, 3rd edition, Chapter 19

Operations:

MakeHeap()

Insert(H, x)

Minimum(H)

ExtractMin(H)

Union(H_1, H_2)

DecreaseKey(H, x, k)

Delete(H, x)

Fibonacci heaps

Operations	Bin. heap worst case	Fib. heap amortized
MakeHeap()	$O(1)$	$O(1)$
Insert(H, x)	$O(\log n)$	$O(1)$
Minimum(H)	$O(1)$	$O(1)$
ExtractMin(H)	$O(\log n)$	$O(\log n)$
Union(H_1, H_2)	$O(n)$	$O(1)$
DecreaseKey(H, x, k)	$O(\log n)$	$O(1)$
Delete(H, x)	$O(\log n)$	$O(\log n)$

More complex structures with worst-case rather than amortized time:

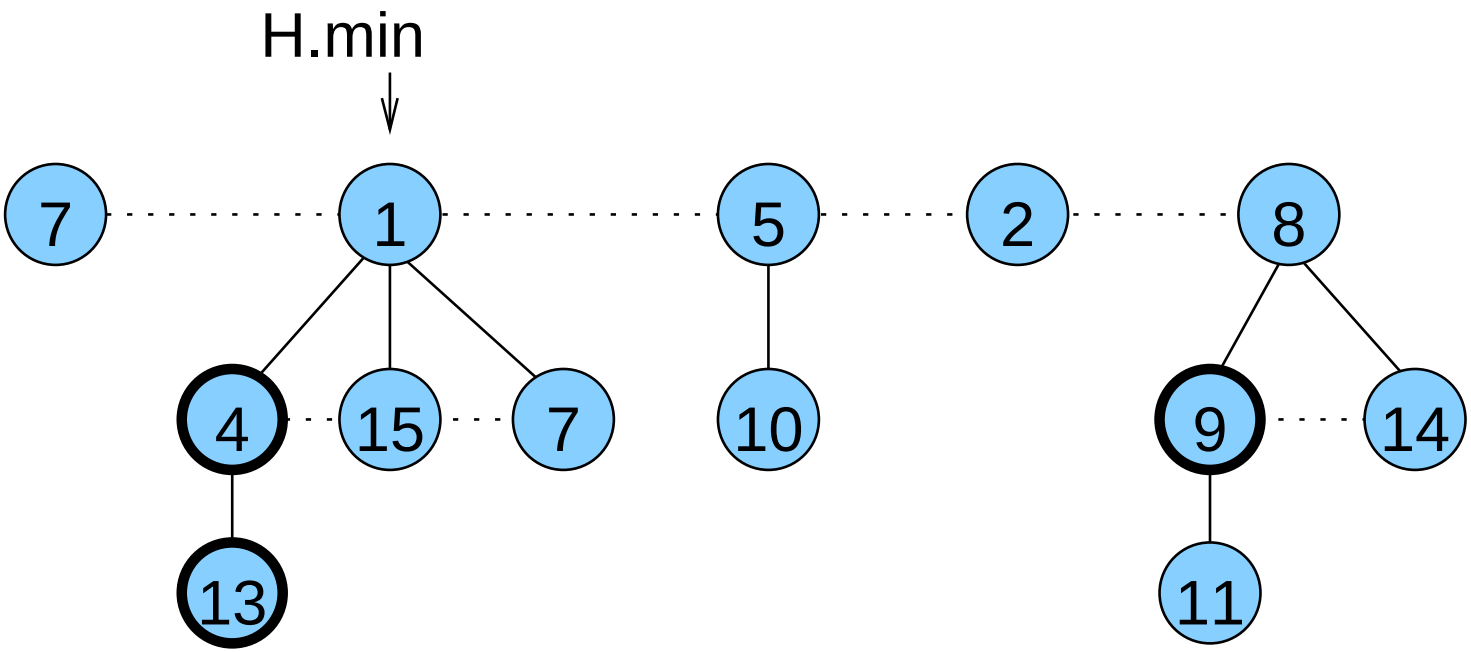
G.S.Brodal: Worst-case efficient priority queues. SODA 1996

Brodal, Lagogiannis, Tarjan: Strict Fibonacci heaps. STOC 2012

Structure of a Fibonacci heap

- Collection of rooted trees, node degrees up to $\mathrm{Deg}(n) = \mathrm{O}(\log n)$
- **Min-heap property:** Key in each non-root node $\geq$ key in its parent
- In each node store: key, other data, parent, first child, next and previous sibling, degree, binary mark
- In each heap: root with min key, number of nodes n, first and last root
- Roots in a heap connected by sibling pointers
- Non-root node is **marked** iff it lost a child since getting a parent
- **Potential function:** number of roots + 2 * number of marked nodes

Example of a Fibonacci heap



Lazy operations with $O(1)$ amortized time

$\Phi =$ number of roots + 2 * number of marked nodes

amortized cost = real cost + $\Phi_{after} - \Phi_{before}$

- MakeHeap: create a new structure
  cost $O(1)$, $\Delta\Phi = 0$
- Insert(H, x): add x as a new root, update $H.n$, $H.min$
  cost $O(1)$, $\Delta\Phi = 1$
- Minimum: return $H.min$
  cost $O(1)$, $\Delta\Phi = 0$
- Union: concatenate root lists, update $H.n$, $H.min$
  cost $O(1)$, $\Delta\Phi = 0$

ExtractMin, $O(\log n)$ amortized

```
1  ExtractMin(H) {
2     z = H.min
3     Add each child of z to root list of H (update its parent)
4     Remove z from root list of H
5     Consolidate(H)
6     return z
7  }
```

Before consolidate, root list may contain up to n roots

Consolidate joins trees until each root has a different degree

Also updates H.min

Let $Deg(n)$ be the maximum degree of a node in a heap of size n

Consolidate

```
1  Consolidate(H) {
2      create array A[0..Deg(H.n)], initialize with null
3      for(each node x in the root list of H) {
4          while(A[x.degree] != null) {
5              y = A[x.degree];
6              A[x.degree] = null;
7              x = HeapLink(H, x, y)
8          }
9          A[x.degree] = x
10     }
11     traverse A, create root list of H, find H.min
12 }
```

HeapLink

```
1  HeapLink(H, x, y) {
2      if(x.key > y key) exchange x and y
3      remove y from root list of H
4      make y a child of x, update x.degree
5      y.mark = false;
6      return x
7  }
```

DecreaseKey, $O(1)$ amortized

```
1  DecreaseKey(H, x, k) {
2      x.key = k
3      y = x.parent
4      if(y!=NULL && x.key < y.key) {
5          Cut(H, x, y)
6          CascadingCut(H, y)
7      }
8      update H.min
9  }
```

Cut: remove x from child list of y, decrement y.degree, add x to the root list, x.parent = null, x.mark = false

Cascading Cut

Node y just lost a child, need to mark it if possible

```
1  CascadingCut(H, y) {
2      z = y.parent
3      if(z != null) {
4          if(y.mark==false) y.mark = true
5          else {
6              Cut(H, y, z);
7              CascadingCut(H, z)
8          }
9      }
10 }
```

Delete $O(\log n)$ amortized

```
1  Delete(H, x) {
2      DecreaseKey(H, x, -infinity)
3      ExtractMin(H)
4  }
```

Fibonacci numbers

$$F_k = \begin{cases} 0 & \text{if } k = 0 \\ 1 & \text{if } k = 1 \\ F_{k-1} + F_{k-2} & \text{if } k \geq 2 \end{cases}$$

Lemma 1: For all $k \geq 0$, we have $F_{k+2} = 1 + \sum_{i=0}^{k} F_i$

Easy proof by induction.

Lemma 2: For all $k \geq 0$, we have $F_{k+2} \geq \phi^k$

where $\phi = (1 + \sqrt{5})/2 = 1.61803\ldots$ is the golden ratio.

Easy proof by induction using the fact that $\phi^2 = \phi + 1$.

Maximum degree of a node

Lemma 3: Let x be a node, and let $y_1, \ldots, y_k$ be its children in the order in which they were linked to x (from earliest). Then $y_j.degree \geq j - 2$ for $j \geq 2$.

Lemma 4: Let x be a node of degree k. Then the size of its subtree is at least $F_{k+2} \geq \phi^k$.

Corollary: The maximum degree $Deg(n)$ of any node in an n-node Fibonacci heap is $O(\log n)$.

Pairing heaps

- Simple and fast in practice, hard to analyze amortized time
- Fredman, Sedgewick, Sleator, Tarjan 1986
- A tree with arbitrary degrees
  – each node pointer to the first child and next sibling
  – delete and decreaseKey also pointer to previous sibling (or parent if first child)
  – min-heap property
- Linking two trees H_1 and H_2 s.t. H_1 has smaller key:
  – make H_2 the first child of H_1 (original first child is sibling of H_2)
  – subtrees thus ordered by age of linking from youngest to oldest
- ExtractMin most complex, other quite lazy