Data structures, Fibonacci heap

Peter Hajnal

Bolyai Institute, University of Szeged, Hungary

2023 fall

Data structures

Data structure: "Structured storage of data".

Each elementary data has an address and parameters. For example: weight: real; key: integer; label: boolean; next, previous: address; ...

The reason to design a data structure is to support the satisfaction of a sequence of requests/services.

Services like `Insert(`$a, \mathcal{S}$`)`, `Delete(`$a, \mathcal{S}$`)`, `DeleteMin(`$\mathcal{S}$`)`, `DecreaseKey(`$a, \mathcal{S}, \delta$`)`.

Services

Let $\mathcal{S}$ be a data structure storing integers, *a* is an elementary data (its key is the stored number).

`DecreaseKey(`$a, \mathcal{S}, \delta$`)` service

To serve the request `DecreaseKey(`$a, \mathcal{S}, \delta$`)` is to modify $\mathcal{S}$ such a way that the key of data *a* is decremented by δ (we assume that $\delta > 0$). The rest of the data is unchanged.

`DeleteMin(`$\mathcal{S}$`)` service

To satisfy `DeleteMin(`$\mathcal{S}$`)` is the modicification of $\mathcal{S}$ by finding the data with minimal key and its deletion from $\mathcal{S}$.

To serve `Insert(`$a, \mathcal{S}$`)`, `Delete(`$a, \mathcal{S}$`)`, `DeleteMin(`$\mathcal{S}$`)`, `DecreaseKey(`$a, \mathcal{S}, \delta$`)` we need to design an algorithm.

Min-heap

In a heap we store integers (keys in data), the data are placed in different vertices of a rooted tree. The notion of heap means the following property:

(H) The key in any data/vertex is at most the keys at its children/descendants.

A heap in the memory I.

A heap $\mathcal{F}$ is an address/pointer, the address of the root node.

A vertex / a data / a key are synonyms.

If our tree is a binary plane tree, then each vertex has a pointer to the left child and right child. A similar solution can be applied if the number of children is bounded in our tree.

We run into a problem if we cannot give a bound on the number of children in our tree, as in our case.

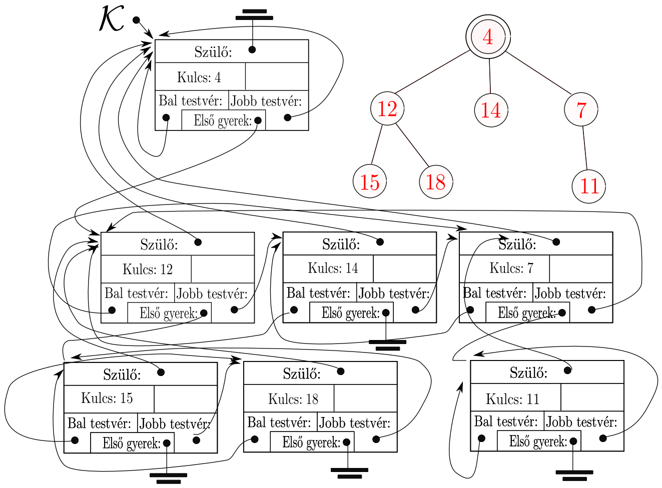
A heap in the memory II.

If the down-degree is not bounded then the memory location allocated to a data contains a pointer to the "first-born-child". This location also contains a pointer to the "next-sibling" and "previous-sibling".

Each node has a memory location containing several pointers.

This part of the memory also contains a key. As we develop our algorithm we will clarify the exact structure of the place assigned to a node. We will see the full picture when our description is complete. It is important to check that the locations assigned to the nodes contain $\mathcal{O}(1)$ components.

A heap in the memory III.: A picture



The structure of a heap $\mathcal{K}$. The keys are the red numbers

Determining the minimal key in a heap

If we have an address/pointer p then $@p$ denotes the data at location p.

$@p[key]$ is the stored numerical value at p.

(H) implies that $@\mathcal{K}[key]$ is the minimal key stored in the heap.

Observation

The minimal key in a heap can be determined in $\mathcal{O}(1)$ steps.

The Fibonacci heap

We describe a simple version of Fibonacci heap. We need to satisfy three kinds of request: `Insert`, `DeleteMin` and `DecreaseKey`.

A Fibonacci heap is actually a system of heaps. The heaps/the roots of the heaps are in a double linked circular chain. Hence every location storing a root has a pointer to the next-heap and to the previous-heap. A Fibonacci heap $\mathcal{F}$ is the link to one of the heaps.

In each heap the keys are arranged according to (H). Specially the root contains the minimal key within a heap. The value of the keys in the different heaps are "independent".

To have a Fibonacci heap we must have
(F) $@\mathcal{F}$[key] is the minimal key in the whole data structure.

Although the different heaps contain independent keys we can find the minimal key of our data easily: in $\mathcal{O}(1)$ steps.

The basic problem

We start with $\mathcal{F}$ and empty data structure.

n denotes the number of requests `Insert`.

In any moment of our algorithm n will be an upper bound on the actual numbers of data in $\mathcal{F}$.

We must efficiently design the structure and the three algorithms to satisfy the requests Insert, DeleteMin, DecreaseKey.

The rank

Definition: Rank of a data

The rank of data a is the number of children of it.

Definition: Rank of a heap

The rank of $\mathcal{H}$ is the rank of the root of $\mathcal{H}$, where $\mathcal{H}$ is a heap in the Fibonacci heap $\mathcal{F}$.

Rank is a dynamically changing parameter as we run our algorithm.

Our goals

Our main goal is that any heap in $\mathcal{F}$ of rank r contains at LEAST α^r data, for suitable constant $\alpha > 1$.

Így seciálisan a fa-kupacok rangja az egész algoritmus során $\mathcal{O}(\log n)$ lesz.

Notation: R

Let R denotes the best upper bound on the ranks of data during the run of our alfgorithm.

If our goal is fulfilled then $R = \mathcal{O}(\log n)$.

Break

Implementing Insert

"Lazy" implementation: We consider the new number as a new one node heap in our system of heaps.

```
$p := @F[NextHeap]$
$@\mathcal{F}[NextHeap] := a$
$@a[NextHeap] := p$
$@a[PreviousHeap] := \mathcal{F}$
$@p[PreviousHeap] := a$
If $@a[Key] < @\mathcal{F}[Key]$ then $\mathcal{F} := a$
```

The cost is $\mathcal{O}(1)$.

Implementing `DeleteMin`: The naive approach

The property (F) guarantees that $\mathcal{F}$ is the address of the data, that we need to delete.

The data, we must delete has a First-Child pointer. Starting from this address we can visit all children of the root. Each child determines a rooted subtree, that is a heap with data stored in it.

We delete

The cost so far is $\mathcal{O}(1)$.

We have to do a final task. We must locate the new minimal key and update the pointer that defines the new Fibonacci heap. We do this by going through the list of roots.

Implementing DeleteMin: Merging phase I.

Merging phase

In the new Fibonacci heap we must browse all roots of the heaps. During this we will do another job: We ensure that the ranks of the heaps in the Fibonacci heap will be different.

During the browsing we have a clean list, where the ranks are different. Furthermore we will have a list leftover heaps.

We define an array $(\rho[i])_{i=0}^{R}$. The elements of the array are pointers. The pointer $\rho[r]$ is the address of the only heap with rank r in the clean list (if there is none, then the value is `nil`).

Implementing DeleteMin: Merging phase II.

At the beginning the clean list is empty, the $(\rho[i])_{i=0}^{R}$ is all-nil, the leftover list contains all the roots.

We take the first leftover heap, $\mathcal{F}$ and assume that its rank is d.

It is valuable to store the rank of a node at the memory location assigned to it. Of course this means an updating task throghout the algorithms. For example we should extend the `Insert` algorithm.

We read $\rho[d]$.

(i) If we see `nil`, then the actual heap is placed from the leftover-list to the clean-list. We update $\rho[d]$.

(ii) If we see an address, then we found the only heap, $\mathcal{F}'$ with rank d in the clean-list. We delete $\mathcal{F}$ from the leftover-list, but we cannot simply place it to the clean-list. We merge the heaps $\mathcal{F}$ and $\mathcal{F}'$.

Implementing DeleteMin: Merging phase III.: Definition of merging

Definition: Merging heaps

From two heaps we construct a new one, that contains the union of the data stored in the two heaps.

The root will be one the original two roots. The root, that contains the smaller key. The other root will be added to it as a child.

The inner structures of the two heaps are preserved.

The rank of the new root is its original rank increased by 1.

Implementing DeleteMin: Merging phase IV.: Recursion

The problem: The clean-list might contain a heap with rank $d + 1$.

We do merging recursively until in the clean-list all keys are different.

Implementing DeleteMin: Merging phase IV.: The cost

Observation

The cost of one merge operation is $\mathcal{O}(1)$.

The total cost of the merge phase is more complicated.

Assume that our initial Fibonacci heap contained k heaps, and after satisfying the `DeleteMin` request the new Fibonacci heap contains k' heap. Let d denote the rank of the original heap, containing the minimum key.

The number of merging is $(k-1)+r-k'$.

The total cost is $\mathcal{O}(k+r)$.

Implementing `DecreaseKey`: The naive phase

$key \leftarrow key - \delta$ is 1 arithmetical operation.

The main problem is that the property (H) must be preserved.

The naive solution: The node, with decreased key generate a rooted subtree, a heap. We cut off this subtree from its parent node. We will consider it as a neww heap in the Fibonacci heap.

We created a new problem: In the Fibonacci heap we might have repeated ranks. We don't care. The next `DeleteMin` service will make some work in that direction.

We must preserve (F). Easy task.

Implementing `DecreaseKey`: Don’t cut too many children

Cutting off children is dangerous: it might leave the rank of the heap untouched, but it might reduce the stored

To the memory location of a node we add a new Boolean component, *Trimmed*: „Did we cut off a child of this node?”. If yes then its value is ‘!’, otherwise ‘$\emptyset$’.

Implementing `DecreaseKey`: Cascading cuts

Cascading cuts

When serving `DecreaseKey` we cut off a node. At this point we check that $\texttt{@}a[\text{Trimmed}] =?$ '!'.

(i) If yes (this cut is not the first one at this node), then we cut it off from its parent too. We recurse (cascade) until we reach the root or we encounter a node, where $\texttt{@}a[\text{Trimmed}] =$'$\emptyset$'.

(ii) If no, i.e. $\texttt{@}a[\text{Trimmed}] =$'$\emptyset$' then cascading stops. We update the Trimmed-value.

Implementing `DecreaseKey`: Final remarks

A cut off/truncation is a local job, its cost is $\mathcal{O}(1)$.

The total cost depends on how many times we perform truncations (the length of the cascading).

Observation

In the Cascading phase we might have many truncations, but only once we set a Trimmed component to be '!'.

Analysis: The main Lemma

Definition

Let $\ell(r)$ be the maximum number that during the algorithm the descendants of a vertex (including itself) of rank r contains at least $\ell(r)$ data.

$\ell(0) = 1$ and $\ell(1) = 2$ are straight forward.

Lemma

Assuming $r \geq 3$, we have

$$\ell(r) \geq \ell(r-2) + \ell(r-3) + \ldots + \ell(0) + 2.$$

Analysis: The proof of the main Lemma I.

We take an arbitrary moment during the run of the Fibonacci heap algorithm, and we consider an arbitrary node v. Let r be the actual number of its children: $v_1, v_2, \dots, v_r$.

The history of v_i can be complex: It might become a child of v by a merge, while serving a `DeleteMin` request. Later it might be cut off from v, and so on. But there must be a last time when it became child of v, and stayed as a child till now. The only reason of becoming a child is performing a merge. The indices of the v_i reflect the last time it became a child of v.

When v_i became a child of v, then v_1, v_2, ..., v_{i-1} were already children of v. At this moment the rank of v was at least $i-1$. A `Merge` caused this event, so at that moment of the merge the rank of v_i was at least $i-1$ too.

Analysis: The proof of the main Lemma II.

At our actual moment v_i is still a child of v. So in the Cascading phase we cut off at most one of its children. The rank of v_i is at least $i-2$ even now.

We can choose a moment and a node v, such that in this moment the rank of v is r and the number of descendants of v is $\ell(r)$.

We have some knowledge about the down-degrees of the children of v. Hence the corresponding descendants can be bounded.

v_2, v_3, …, v_r and their descendants give

$$\ell(0)+\ell(1)+\ldots+\ell(r-2)$$

nodes. v and v_1 two further nodes.

The claim is proven.

Analysis: Fibonacci numbers

Definition: Fibonacci sequence/Fibonacci numbers

Let $F_0 = 1, F_1 = 2$, furthermore $F_n = F_{n-1} + F_{n-2}$ if $n \geq 2$.

By induction one obtains: $F_n \geq \left(\frac{1+\sqrt{5}}{2}\right)^n$.

Again by induction:

$$\ell(r) \geq F_r \geq \left(\frac{1+\sqrt{5}}{2}\right)^r .$$

Our promise is proven. So now on we know that $R = \mathcal{O}(\log n)$.

We also know why is that a data structure from the XXth century got its name from a mathematician who was born in XIth century.

Amortized analyis: `Insert`

The real cost of `Insert` is $\mathcal{O}(1)$.

We define the amortized cost the previous amount plus a "merge deposit". We think this extra as an amount placed next to node. We might ude to perform a merging in the future and use this money to pay for it.

(P): Promise

Any moment of the run the roots of the heaps in the Fibonacci heap have a "merge deposit".

Theorem

The amortized cost of `Insert` operation is $\mathcal{O}(1)$.

Amortized analyis: `DeleteMin`

Theorem

The amortized cost of an `DeleteMin` operation is $\mathcal{O}(R) = \mathcal{O}(\log n)$.

The list of children of the deleted node should be added to the list of heaps. The cost is $\mathcal{O}(1)$.

The children become roots of heaps. To keep (P) we put a "merge deposit" to them. Its cost is $\mathcal{O}(R)$.

We have an initial array of length R. To form this costs $\mathcal{O}(R)$.

Updating the array can be managed from the "merge deposits".

Amortized analyis: `DecreaseKey`

The naive cost of `DecreaseKey` is $\mathcal{O}(1)$. We call it "cut-cost".

We add two "merge-deposits" and one "cut-deposit" to the cost so far. This way we get the amortized cost.

The cut puts a subtree (the generated subtree by the node where the decrement happened) among the heaps. That require a "merge deposit" at the root. This can be paid by the first "merge-deposit".

Remember that the end of the Cascading a Trimmed component might be set '!'. That means a possible cut in the future. The "cut-deposit" and the second "merge-deposit" will be placed next to the node with '!'.

We maintain the property that nodes with '!' always have "cut-" and "merge-deposit" placed to it. The cost of the Cascading can be paid from deposits.

Amortized analyis: DecreaseKey: Summary

Theorem

The amortized cost of DecreaseKey is $\mathcal{O}(1)$.

The Theorem

Theorem

We start with an empty Fibonacci heap, and satisfy a sequence of requests consisting of n `Insert`, m `DeleteMin` and d `DecreaseKey`. The the cost of our algorithm is

$$\mathcal{O}(n + m \cdot \log n + d).$$

Break

The shortest path problem

The input of the shortest path problem (G, ℓ, s), where G is a directed graph, ℓ is a length function on $E(G)$, and finally s is special node called source.

The length function on the edge set can be extended to a length function on walks. Based on that we can define the distance from vertex x to vertex y.

The output the problem is for each node $v \in V(G)$ determining the length of the shortest path from s to v.

Dijkstra's algorithm

Our description of the algorithm was maintaining a vertex set $\widehat{S}$ with a number/key assigned to each element of $\widehat{S}$.

We choose the element m with minimal key from $\widehat{S}$ and deleted them. // We knew the distance from s.

We update of the keys in $\widehat{S}$: Some keys will be decreased.

The out-neighbors of m (from the leftover set of vertices) are inserted into $\widehat{S}$ with suitable key.

We do this until $\widehat{S}$ becomes empty.

Implementation of Dijkstra's algorithm based on Fibonacci heap

Our description emphasizes the fact that Fibonacci heaps can be used it to design an algorithm the performs Dijkstra algorithm.

During the run of Dijkstra's algorithm we have $|V|$ `Insert`, $|E|$ `DecreaseKey` and $|V|$ `DeleteMin` operations.

Theorem

Dijkstra's algorithms can be implemented in

$$\mathcal{O}(|V| \log |V| + |E|)$$

time.

This is the end!

Thank you for your attention!