Principles of Algorithmic Techniques
CS-E3190

Pekka Orponen
Department of Computer Science
Aalto University

Autumn 2017
Lecture 16: Fundamental data structures

- Data types and data structures
- Priority queues and heaps
- Disjoint sets and union-find trees
16.1 Data types and data structures

- It is useful to separate abstractions and implementations.
- In programming, an algorithm is an abstract (general) method, whereas a program in some programming language is its concrete implementation.
- In data structures, an (abstract) data type is some family of mathematical structures, with associated elementary operations transforming one structure into another. A given data type can often be implemented using several different concrete data structures.
- E.g. the abstract data type of a sequence of elements can be implemented concretely using arrays, linked lists, etc.
- Different implementations typically have different advantages and disadvantages w.r.t. storage space requirements, efficiency of elementary operations, simplicity of coding, etc.
16.2 Priority queues and heaps

- A priority queue is a data type representing the storage of a finite set $H$ of elements, with associated key values from some linearly ordered universe (integers, strings, etc.).
- The key values may be thought to represent the “priorities” of the stored elements: the lower the key, the more “important” or “timely” the element.
- A priority queue $H$ is only accessed via the basic operations (“methods”):
  - $\text{INSERT}(H, (u, x))$: insert element $u$ with associated key value $x$ into $H$.
  - $\text{DELETEMIN}(H)$: extract the element $u$ with the current smallest key value $x$ from $H$; return pair $(u, x)$.
  - $\text{DECREASEKEY}(H, (u, x))$: decrease the key value ("priority") of element $u$ to $x$.
- Priority queues are used e.g. in the designs of Dijkstra’s and Prim’s algorithms.
Implementations of priority queues

- The set of basic operations for a priority queue may be implemented by many different concrete data structures.
- Different implementations result in different time complexities for the basic operations.
- E.g. simple array implementation:
  - Assume the base set of all possible elements \( u \) is some \( [N] = \{1, \ldots, N\} \).
  - Represent a priority queue by an array \( H[1 \ldots N] \) s.th.
    \[
    \begin{cases} 
      H[u] = x, & \text{if } u \in H \text{ with key } x, \\
      H[u] = \infty, & \text{if } u \notin H. 
    \end{cases}
    \]
  - Complexities of operations:
    \[
    \begin{align*}
    \text{INSERT}(H, (u, x)) & : \mathcal{O}(1) \quad \text{Good} \\
    \text{DECREASE\textsc{key}}(H, (u, x)) & : \mathcal{O}(1) \quad \text{Good} \\
    \text{DELETE\textsc{min}}(H) & : \mathcal{O}(N) \quad \text{Bad}
    \end{align*}
    \]
Heaps

- A more balanced implementation is achieved by the heap data structure.
- This is simply a complete binary tree\(^1\) storing the \((u, x)\)-pairs of a priority queue \(H\) with the condition:
  - The key value at any vertex is less than or equal to the key value at any of its children.
- An example heap with 10 elements (only key values shown):

\(^1\) That is, a binary tree with each level full except possibly the lowest one.
Heap implementation of priority queues (1/4)

- The simple (and clever) partial ordering condition of a heap facilitates very efficient implementations of the priority queue operations.
- Operation $\text{INSERT}(u, x)$ is implemented by placing the key $x$ for element $u$ at the bottom of the tree and letting it “bubble up”. That is, if $x$ is located at a vertex whose parent has a larger key value, swap the two and repeat.
Heap implementation of priority queues (2/4)

- Operation $\text{DECREASE\textsc{Key}}(u, x)$ is implemented similarly to Insert, except that now the old key $y$ for element $u$ is already located in the tree.
- Thus key $y$ needs to be replaced by key $x < y$ and the “bubble up” process then initiated correspondingly.\(^2\)
- Since a complete binary tree of $n$ vertices has height $\lfloor \log_2 n \rfloor$, and each swap operation takes $O(1)$ time, both INSERT and DECREASE\textsc{Key} applied to an $n$-node heap have time complexity $O(\log n)$.

\(^2\) Note that for this one needs some external structure, e.g. an array of pointers, linking elements $u$ to their key values in the tree.
Heap implementation of priority queues (3/4)

- For `DELETEMIN`, return the \((u, x)\) pair associated to the root of the tree. Take the last element from the tree (rightmost position in the bottom row), replace the root by this and let the new root key \(x'\) “sift down”: if a key is larger than at least one of its children, swap it with the smaller child and repeat.

- Clearly also this operation takes only \(O(\log n)\) time.
Heap implementation of priority queues (4/4)

- Using heaps one thus gets a uniform cost of $O(\log n)$ per operation for each of the priority queue operations.\(^3\)

- A further note on implementation: a complete binary tree $T$ of $n$ vertices can be conveniently packed into an array $H[1 \ldots n]$ by just listing the vertex keys of $T$ in order, row by row, as elements of $H$.

- Then the children of a vertex mapped to $H[j]$ are located at $H[2j]$ and $H[2j + 1]$, unless the respective indices are beyond $n$. Correspondingly: if $j > 1$, then the parent of vertex $H[j]$ is located at $H[\lfloor j/2 \rfloor]$.

\(^3\)With the more advanced structure of Fibonacci heaps one can even push the cost of INSERT’s and DECREASEKEY’s to $O(1)$ per operation, when amortised over long operation sequences.
Heapsort

The heap data structure can also be used as the basis of an elegant sorting method.

Algorithm 1: The heapsort algorithm

1. function HEAPSORT (A[1 ... n]);
   
   Input: Integer array A[1 ... n]
   Output: Array A, with same items in increasing order.

2. Let H be a heap structure of n elements;

3. for i ← 1 to n do INSERT(H, A[i]);

4. for i ← 1 to n do A[i] ← DELETEMIN(H);

For an n-element heap, each of the elementary operations takes time $O(\log n)$, hence the algorithm sorts an n-element array in time $O(n \log n)$. 
Another fundamental data type are disjoint sets.
Here the basic configuration is a partition of a finite base set of elements into a family of disjoint classes.
The state of the partition can be queried, and it can be made coarser by merging classes together, but no other operations are supported.
The basic operations in this data type are the following:
- \( \text{MAKESET}(x) \): Create and name a singleton set containing only element \( x \).
- \( \text{FIND}(x) \): Return the name of the set into which element \( x \) currently belongs.
- \( \text{UNION}(x, y) \): Merge the (disjoint) sets containing elements \( x \) and \( y \) into one. The name of the new set is either one of the names of the old sets.
Disjoint sets are used e.g. in the design of Kruskal’s algorithm.
An efficient representation of such a family of disjoint sets, supporting the operations of interest, is in terms of a forest of rooted trees.

The elements in each set are associated to the vertices of a tree, with links pointing towards the root. The root of the tree is used as the tree’s (set’s) label or name.

A forest representing disjoint sets $E : \{B, E\}$ and $H : \{A, C, D, F, G, H\}$ would be the following.

![Tree representation](attachment://tree_representation.png)
Tree implementation of disjoint sets (2/4)

The basic disjoint set operations are easy to implement in this data structure. Associate to each element $x$ in the base set a parent pointer $\pi(x)$.

Algorithm 2: Basic disjoint set operations

1. function `MAKESET(x)`;
2. $\pi(x) \leftarrow x$;
3. function `FIND(x)`;
4. while $x \neq \pi(x)$ do $x \leftarrow \pi(x)$;
5. return $x$;
6. function `UNION(x, y)`;
7. $r_x \leftarrow FIND(x); r_y \leftarrow FIND(y)$;
8. if $r_x \neq r_y$ then
   9. Set either $\pi(r_y) \leftarrow r_x$ or $\pi(r_x) \leftarrow r_y$;
10. end
Tree implementation of disjoint sets (3/4)

- In this implementation, \texttt{MAKESET} is a constant-time operation, but the complexities of \texttt{FIND} and \texttt{UNION} depend on the heights of the trees at hand.

- In order to keep the trees balanced, associate further to each element \( x \) its \textit{rank} \( \text{rk}(x) \), which tentatively indicates the height of the subtree currently below it.

\begin{algorithm}
\textbf{Algorithm 3:} Disjoint set operations with rank (1/2)

\begin{enumerate}
\item \textbf{function} \texttt{MAKESET}(x);
\item \( \pi(x) \leftarrow x \);
\item \( \text{rk}(x) \leftarrow 0 \);
\item \textbf{function} \texttt{FIND}(x);
\item \textbf{while} \( x \neq \pi(x) \) \textbf{do} \( x \leftarrow \pi(x) \);
\item \textbf{return} \( x \)
\end{enumerate}
\end{algorithm}
Tree implementation of disjoint sets (4/4)

▶ To maintain tree balance, in UNION operations the shorter tree is always made a subtree of the taller one.
▶ Note that then the tree height increases only if the two trees being merged are equally tall.

Algorithm 4: Disjoint set operations with rank (2/2)

1. function UNION(x, y);
2. $r_x \leftarrow$ FIND(x); $r_y \leftarrow$ FIND(y);
3. if $r_x = r_y$ then return;
4. if $\text{rk}(r_x) > \text{rk}(r_y)$ then
5.     $\pi(r_y) \leftarrow r_x$
6. end
7. else
8.     $\pi(r_x) \leftarrow r_y$;
9.     if $\text{rk}(r_x) = \text{rk}(r_y)$ then $\text{rk}(r_y) = \text{rk}(r_y) + 1$;
10. end
Example of a union-find operation sequence (1/2)

Superscript on element names indicate their current rank.

After $\text{MAKESET}(A), \text{MAKESET}(B), \ldots, \text{MAKESET}(G)$:

\[
\begin{array}{ccccccc}
A^0 & B^0 & C^0 & D^0 & E^0 & F^0 & G^0 \\
\end{array}
\]

After $\text{UNION}(A, D), \text{UNION}(B, E), \text{UNION}(C, F)$:
Example of a union-find operation sequence (2/2)

After \texttt{UNION}(C, G), \texttt{UNION}(E, A):

After \texttt{UNION}(B, G):
Complexity of union-find operations (1/2)

The vertex ranks of a union-find forest have the following simple properties:

1. For any vertex $x$, $\text{rk}(x) < \text{rk}(\pi(x))$.
   - By design of the rank updates. Also note that the rank of a vertex equals the height of the subtree rooted at that vertex.

2. Any root vertex of rank $k$ has at least $2^k$ vertices in its tree.
   - By induction on the rank updates. A root vertex of rank $k$ is created by the merger of two trees of rank $k - 1$.

3. In a forest of $n$ vertices, at most $n/2^k$ vertices have rank $k$.
   - When a vertex becomes a nonroot, its rank no longer changes. Thus by Property 2 any rank-$k$ vertex has at least $2^k$ descendants, and by Property 1 no rank-$k$ vertex is a descendant of another.
Property 3 of vertex ranks implies that the maximum rank achievable in a forest of \(n\) vertices is \(\log_2 n\).

Therefore, by Property 1, all trees in such a forest have height \(\leq \log_2 n\), and hence all FIND and UNION operations have complexity \(O(\log n)\).
Path compression (1/2)

- Time complexity $\mathcal{O}(\log n)$ can be seen to be tight in the worst case for a given FIND or UNION operation.
  - E.g. the scheme in the previous example constructs a union-find tree of $n$ vertices and depth $\Omega(\log n)$. A FIND operation to the leaf of such a tree has cost $\Omega(\log n)$.
- Surprisingly, the FIND and UNION operations can be made almost constant\(^4\) when amortised over full operation sequences.
  - Intuitively, thus, the construction of a tree enabling a FIND of cost $T$ time units requires an operation sequence of length almost $T$.

\(^4\)But not quite!
Path compression (2/2)

- The simple idea facilitating this improvement is path compression of search paths in connection with the FIND operations.
- In path compression, the search path associated to operation FIND\((x)\) is “squashed” by making all of the vertices \(y\) encountered during the traversal from vertex \(x\) to the respective root \(r_x\) into direct descendants of \(r_x\).
- This is achieved by simply updating the parent pointer \(\pi(y)\) of each vertex \(y\) along the path to point directly to \(r_x\).

Algorithm 5: FIND with path compression (using recursion)

1. function FIND\((x)\);
2. if \(x \neq \pi(x)\) then \(\pi(x) \leftarrow\) FIND\((\pi(x))\);
3. return \(\pi(x)\)
Example of path compression

Operation $\text{FIND}(I)$ followed by $\text{FIND}(K)$:

To illustrate the effect of path compression, consider the path:

```
A^3 -> B^0 -> C^1 -> D^0
```

After applying find on $B^0$, the path becomes:

```
A^3 -> B^0 -> C^1
```

And finally, after applying find on $C^1$, the path is simplified to:

```
A^3
```

In practice, applying $\text{FIND}(I)$ followed by $\text{FIND}(K)$ brings it down to 1 (or below 1).

For instance, $\log^* 1000 = 4$ since $\log \log \log \log 1000 \leq 1$.

In a sequence of find operations, some may take longer than others. We'll bound the overall running time using some creative accounting. Specifically, we will give each node a certain amount of pocket money, such that the total money doled out is at most $n \log^* n$ dollars.

We will then show that each find takes $O(\log^* n)$ steps, plus some additional amount of time that can be "paid for" using the pocket money of the nodes involved—one dollar per unit of time. Thus the overall time for $m$ finds is $O(m \log^* n)$ plus at most $O(n \log^* n)$.

In more detail, a node receives its allowance as soon as it ceases to be a root, at which point its rank is fixed. If this rank lies in the interval $\{k+1, \ldots, 2k\}$, the node receives $2k$ dollars.

By Property 3, the number of nodes with rank $> k$ is bounded by $n^{2^k} + n^{2^k} + \cdots \leq n^{2^k}$.

Therefore the total money given to nodes in this particular interval is at most $n$ dollars, and since there are $\log^* n$ intervals, the total money disbursed to all nodes is $\leq n \log^* n$.
Complexity effect of path compression

- Path compression does not affect directly the worst-case time complexity of any individual UNION or FIND operation.

- However, one can show that an arbitrary sequence of $n$ UNION’s and FIND’s (with path compression) takes time at most $O(n \log^* n)$, where

$$\log^* n = \min\{t \mid \underbrace{2 \cdot 2 \cdots 2}_{t} \geq n\}.$$ 

- Thus, in an amortised sense, i.e. averaged over operations in an arbitrary sequence of length $n$, each operation takes only time $O(\log^* n)$.

- Note that $\log^* n \leq 5$ for all practical purposes ($n \leq 2^{65536}$).
Proof of complexity bound (1/3)

**Theorem.** Using path compression, any sequence $P$ of $n$ interleaved UNION’s and FIND’s takes time at most $O(n \log^* n)$.

**Proof.** Let $\mathcal{F}$ be the union-find forest that would be created by sequence $P$ without path compression.

Since each UNION comprises two FIND’s plus some constant-time pointer updates, we may w.l.o.g. include these FIND’s in our sequence $P$ and ignore the other costs. (The length of $P$ increases to at most $2n$ and the total cost of the pointer updates is $O(n)$, neither of which changes the result.)

Even though path-compression changes the structure of the search trees in $\mathcal{F}$, it does not affect the rank values assigned to the vertices. In particular Properties 1–3 of the rank values still hold.
Proof of complexity bound (2/3)

Now group the vertices $x$ in $\mathcal{F}$ into rank-intervals $R_t$, $t = 0, \ldots, \log^* n$, of exponentially increasing length, so that

$$R_t = \{k + 1, \ldots, 2^k\}, \text{ where } k = k_t = 2^{2^{\cdot^{(t \text{ times})}}}. $$

Let the sequence of FIND operations in $P$ be $F_1, \ldots, F_n$. The cost of $P$ is proportional to the total number of pointers $x \rightarrow \pi(x)$ followed in these.

For a given operation $F_i$, divide the pointer count between $F_i$ and the vertices $x$ traversed as follows:

- Charge $F_i$ for the last pointer (leading to the root of the current tree) and for all the pointers $x \rightarrow \pi(x)$ where $x$ and $\pi(x)$ are in different rank-intervals.
- Charge each $x$ for the pointer $x \rightarrow \pi(x)$ when $x$ and $\pi(x)$ are in the same rank-interval, and $\pi(x)$ is not the root.

Altogether,

$$c(P) = \sum_i c(F_i) + \sum_x c(x).$$
Proof of complexity bound (3/3)

Clearly, for each $F_i$, $c(F_i) \leq 1 + \log^* n$, and so

$$\sum_{i=1}^{n} c(F_i) \leq n + n \log^* n.$$ 

On the other hand, whenever a vertex $x$ is charged for a pointer, its parent changes to be of higher rank (path compression + Property 1). Thus, any $x$ belonging to rank-interval $R_t = \{k + 1, \ldots, 2^k\}$ will accumulate a total charge of at most $2^k$ before its parent moves to a different interval.

By Property 3, $|R_k| \leq \frac{n}{2^{k+1}} + \frac{n}{2^{k+2}} + \cdots \leq \frac{n}{2^k}$, and so:

$$\sum_{x} c(x) = \sum_{t=0}^{\log^* n} \sum_{x \in R_t} c(x) \leq \sum_{t=0}^{\log^* n} \frac{n}{2^{k_t}} \cdot 2^{k_t} = (1 + \log^* n) \cdot n.$$ 

The result follows.