Priority Queue / Heap

- Stores \((key, data)\) pairs (like dictionary)
- But, different set of operations:

  - **Initialize-Heap**: creates new empty heap
  - **Is-Empty**: returns true if heap is empty
  - **Insert\((key, data)\)**: inserts \((key, data)\)-pair, returns pointer to entry
  - **Get-Min**: returns \((key, data)\)-pair with minimum \(key\)
  - **Delete-Min**: deletes minimum \((key, data)\)-pair
  - **Decrease-Key\((entry, newkey)\)**: decreases \(key\) of \(entry\) to \(newkey\)
  - **Merge**: merges two heaps into one
Implementation of Dijkstra’s Algorithm

Store nodes in a priority queue, use $d(s, v)$ as keys: $G = (V, E)$

1. Initialize $d(s, s) = 0$ and $d(s, v) = \infty$ for all $v \neq s$
2. All nodes are unmarked
   - Initialize, insert all nodes $v$ with init.key $\infty$
3. Get unmarked node $u$ which minimizes $d(s, u)$:
   - get-min
4. Mark node $u$
   - delete-min
5. For all $e = \{u, v\} \in E$, $d(s, v) = \min\{d(s, v), d(s, u) + w(e)\}$
   - for all neighbors of $u$: decrease-key
6. Until all nodes are marked
   - is-empty
Analysis

Number of priority queue operations for Dijkstra:

• Initialize-Heap: \(1\)

• Is-Empty: \(|V|\)

• Insert: \(|V|\)

• Get-Min: \(|V|\)

• Delete-Min: \(|V|\)

• Decrease-Key: \(|E|\)

• Merge: \(0\)
Priority Queue Implementation

Implementation as min-heap:

- Complete binary tree, e.g., stored in an array

- **Initialize-Heap**: $O(1)$
- **Is-Empty**: $O(1)$
- **Insert**: $O(\log n)$
- **Get-Min**: $O(1)$
- **Delete-Min**: $O(\log n)$
- **Decrease-Key**: $O(\log n)$
- **Merge** (heaps of size $m$ and $n$, $m \leq n$): $O(m \log n)$
Better Implementation

• Can we do better?

• Cost of Dijkstra with complete binary min-heap implementation:

\[ O(|E| \log|V|) \]

• Can be improved if we can make decrease-key cheaper...

• Cost of merging two heaps is expensive

• We will get there in two steps:

  Binomial heap \rightarrow \text{Fibonacci heap}
Definition: Binomial Tree

Binomial tree $B_n$ of order $n$ ($n \geq 0$):
Binomial Trees

\[ B_0 \quad B_1 \quad B_2 \quad B_3 \]
Properties

1. Tree $B_n$ has $2^n$ nodes
   - induction on $n$: $n=0$ ✓
   - step: $B_{n-1}$
   - $|B_n| = 2|B_{n-1}|

2. Height of tree $B_n$ is $n$
   - $n=0$ ✓
   - step: $n-1$
   - $n-1$

3. Root degree of $B_n$ is $n$
   - $n=0$ ✓
   - step: $B_{n-1}$

4. In $B_n$, there are exactly $\binom{n}{i}$ nodes at depth $i$
Binomial Coefficients

- Binomial coefficient:
  \( \binom{n}{k} \): # of \( k \) element subsets of a set of size \( n \)

- Property: \( \binom{n}{k} = \binom{n-1}{k-1} + \binom{n-1}{k} \)

Pascal triangle:
Number of Nodes at Depth $i$ in $B_n$

Claim: In $B_n$, there are exactly $\binom{n}{i}$ nodes at depth $i$
Binomial Heap

• Keys are stored in nodes of binomial trees of different order

  **n nodes:** there is a binomial tree $B_i$ of order $i$ iff bit $i$ of base-2 representation of $n$ is 1.

  $$n = 21 = 2^4 + 2^2 + 2^0 = (10101)_2 \quad \left| B_i \right| = 2^i$$

• **Min-Heap Property:**

  Key of node $v \leq$ keys of all nodes in sub-tree of $v$
Example

- 10 keys: \{2, 5, 8, 9, 12, 14, 17, 18, 20, 22, 25\}

- Binary representation of \( n \): \((11)_2 = 1011\)
  \(\rightarrow\) trees \(B_0, B_1,\) and \(B_3\) present
Child-Sibling Representation

Structure of a node:

<table>
<thead>
<tr>
<th></th>
<th>parent</th>
<th>key</th>
<th>degree</th>
</tr>
</thead>
<tbody>
<tr>
<td>child</td>
<td></td>
<td></td>
<td>sibling</td>
</tr>
</tbody>
</table>

Diagram showing the structure of nodes in a child-sibling representation.
Link Operation

- Unite two binomial trees of the same order to one tree:
  \[ B_n \oplus B_n \Rightarrow B_{n+1} \]

- Time: \( O(1) \)
Merge Operation

Merging two binomial heaps:

• For \( i = 0, 1, \ldots, \log n \):
  If there are 2 or 3 binomial trees \( B_i \): apply link operation to merge 2 trees into one binomial tree \( B_{i+1} \)

\[
\begin{align*}
Q_1 & \rightarrow B_0 \rightarrow B_5 \rightarrow B_6 \rightarrow B_9 \rightarrow B_{10} \rightarrow B_{11} \\
Q_2 & \rightarrow B_0 \rightarrow B_5 \rightarrow B_8 \rightarrow B_{10} \rightarrow B_{11} \\
Q_1 \cup Q_2 & \rightarrow B_1 \rightarrow B_7 \rightarrow B_8 \rightarrow B_9 \rightarrow B_{11} \rightarrow B_{12}
\end{align*}
\]

Time: \( O(\log n) \)
Example

\[ \begin{align*}
9 & \quad 13 \\
12 & \quad 18 \\
14 & \quad 20 \\
5 & \quad 2 \\
2 & \quad 17 \\
22 & \\
25 & \\
\end{align*} \]

Diagram
Operations

**Initialize**: create empty list of trees

**Get minimum** of queue: time $O(1)$ (if we maintain a pointer)

**Decrease-key** at node $v$:
- Set key of node $v$ to new key
- Swap with parent until min-heap property is restored
- Time: $O(\log n)$

**Insert** key $x$ into queue $Q$:
1. Create queue $Q'$ of size 1 containing only $x$
2. Merge $Q$ and $Q'$
- Time for insert: $O(\log n)$
Operations

Delete-Min Operation:

1. Find tree $B_i$ with minimum root $r$
   
2. Remove $B_i$ from queue $Q \rightarrow$ queue $Q'$

3. Children of $r$ form new queue $Q''$

4. Merge queues $Q'$ and $Q''$

- Overall time: $O(\log n)$
Delete-Min Example

Algorithm Theory, WS 2012/13

Fabian Kuhn
Complexities Binomial Heap

- Initialize-Heap: $O(1)$
- Is-Empty: $O(1)$
- Insert: $O(\log n)$
- Get-Min: $O(1)$
- Delete-Min: $O(\log n)$
- Decrease-Key: $O(\log n)$
- Merge (heaps of size $m$ and $n$, $m \leq n$): $O(\log n)$

Dijkstra is still $O(\text{IE}\log\text{IVI})$
Can We Do Better?

• Binomial heap: 
  insert, delete-min, and decrease-key cost \( O(\log n) \)

• One of the operations insert or delete-min must cost \( \Omega(\log n) \):
  – Heap-Sort: 
    Insert \( n \) elements into heap, then take out the minimum \( n \) times
  – (Comparison-based) sorting costs at least \( \Omega(n \log n) \).

• But maybe we can improve decrease-key and one of the other two operations?

• Structure of binomial heap is not flexible:
  – Simplifies analysis, allows to get strong worst-case bounds
  – But, operations almost inherently need at least logarithmic time
Fibonacci Heaps

Lacy-merge variant of binomial heaps:
• Do not merge trees as long as possible...

Structure:
A Fibonacci heap \( H \) consists of a collection of trees satisfying the min-heap property.

Variables:
• \( H.min \): root of the tree containing the (a) minimum key
• \( H.rootlist \): circular, doubly linked, unordered list containing the roots of all trees
• \( H.size \): number of nodes currently in \( H \)
Trees in Fibonacci Heaps

Structure of a single node $v$:

- $v.\text{child}$: points to circular, doubly linked and unordered list of the children of $v$
- $v.\text{left}, v.\text{right}$: pointers to siblings (in doubly linked list)
- $v.\text{mark}$: will be used later...

Advantages of circular, doubly linked lists:
- Deleting an element takes constant time
- Concatenating two lists takes constant time
Example

Figure: Cormen et al., Introduction to Algorithms
Simple (Lazy) Operations

**Initialize-Heap** $H$:
- $H.\text{rootlist} := H.\text{min} := \text{null}$

**Merge** heaps $H$ and $H'$:
- concatenate root lists
- update $H.\text{min}$

$\mathcal{O}(1)$ time

**Insert** element $e$ into $H$:
- create new one-node tree containing $e \rightarrow H'$
- merge heaps $H$ and $H'$

**Get minimum** element of $H$:
- return $H.\text{min}$
Operation Delete-Min

Delete the node with minimum key from $H$ and return its element:

1. $m := H.min$;
2. if $H.size > 0$ then
3. remove $H.min$ from $H.rootlist$;  // delete min
4. add $H.min.child$ (list) to $H.rootlist$  // merge 2 heaps
5. $H.Consolidate()$;
   // Repeatedly merge nodes with equal degree in the root list
   // until degrees of nodes in the root list are distinct.
   // Determine the element with minimum key
6. return $m$
Rank and Maximum Degree

Ranks of nodes, trees, heap:

Node $v$:
- $\text{rank}(v)$: degree of $v$

Tree $T$:
- $\text{rank}(T)$: rank (degree) of root node of $T$

Heap $H$:
- $\text{rank}(H)$: maximum degree of any node in $H$

Assumption ($n$: number of nodes in $H$):
- $\text{rank}(H) \leq D(n)$
  - for a known function $D(n)$
Merging Two Trees

**Given:** Heap-ordered trees $T, T'$ with $\text{rank}(T) = \text{rank}(T')$

- Assume: min-key of $T \leq$ min-key of $T'$

**Operation $\text{link}(T, T')$:**

- Removes tree $T'$ from root list and adds $T'$ to child list of $T$
- $\text{rank}(T') := \text{rank}(T) + 1$
- $T'. \text{mark} := \text{false}$
Consolidation of Root List

Array $A$ pointing to find roots with the same rank:

\[ \begin{array}{c|c|c|c} & 0 & 1 & 2 \\ \hline D(n) & & & \ldots \end{array} \]

**Consolidate:**

1. for $i := 0$ to $D(n)$ do $A[i] := \text{null}$;
2. while $H\.rootlist \neq \text{null}$ do
3. $T := \text{"delete and return first element of } H\.rootlist"$
4. while $A[\text{rank}(T)] \neq \text{null}$ do
5. $T' := A[\text{rank}(T)]$;
6. $A[\text{rank}(T)] := \text{null}$;
7. $T := \text{link}(T, T')$
8. $A[\text{rank}(T)] := T$
9. Create new $H\.rootlist$ and $H\.min$

**Time:** $O(|H\.rootlist| + D(n))$
Consolidate Example

link

Algorithm Theory, WS 2012/13
Fabian Kuhn
Consolidate Example

Algorithm Theory, WS 2012/13

Fabian Kuhn
Consolidate Example
Consolidate Example

\[
\begin{array}{c}
5 \rightarrow 1 \\
9 \rightarrow 25 \rightarrow 31 \rightarrow 13 \\
14 \rightarrow 12 \rightarrow 18 \rightarrow 3 \\
20 \rightarrow 19 \rightarrow 7 \\
\end{array}
\]
Consolidate Example

link

Algorithm Theory, WS 2012/13
Fabian Kuhn
Consolidate Example

- **link**
- **length of root list**
- $O(D(w))$

Algorithm Theory, WS 2012/13  Fabian Kuhn 35
Operation Decrease-Key

Decrease-Key(v, x): (decrease key of node v to new value x)

1. if x ≥ v.key then return;
2. v.key := x; update H.min;
3. if v ∈ H.rootlist ∨ x ≥ v.parent.key then return
4. repeat
   5. parent := v.parent;
   6. H.cut(v);
   7. v := parent;
5. until ¬(v.mark) ∨ v ∈ H.rootlist;
6. if v ∉ H.rootlist then v.mark := true;
Operation Cut(\(v\))

Operation \(H.\)cut(\(v\)):
- Cuts \(v\)'s sub-tree from its parent and adds \(v\) to rootlist

1. **if** \(v \notin H.\)rootlist **then**
2. // cut the link between \(v\) and its parent
3. \(\text{rank}(v.\text{parent}) := \text{rank}(v.\text{parent}) - 1;\)
4. remove \(v\) from \(v.\text{parent}.\)child (list)
5. \(v.\text{parent} := \text{null};\)
6. add \(v\) to \(H.\)rootlist

---

**Diagram:**

Before cut:
- Tree with nodes 1, 2, 15, 25, 13, 3, 8, 31, 19, 7
- Node \(v\) is highlighted

After cut:
- Tree with nodes 1, 3, 2, 15, 25, 13, 19, 7, 8
- Node \(v\) is added to rootlist
Decrease-Key Example

• Green nodes are marked

\[ \text{Decrease-Key}(v, 8) \]
Fibonacci Heap Marks

History of a node $v$:

- $v$ is being linked to a node $\rightarrow v.\text{mark} := \text{false}$

- A child of $v$ is cut $\rightarrow v.\text{mark} := \text{true}$

- A second child of $v$ is cut $\rightarrow H.\text{cut}(v)$

- Hence, the boolean value $v.\text{mark}$ indicates whether node $v$ has lost a child since the last time $v$ was made the child of another node.
Cost of Delete-Min & Decrease-Key

Delete-Min:
1. Delete min. root \( r \) and add \( r.\)child to \( H.\)rootlist
   time: \( O(1) \)
2. Consolidate \( H.\)rootlist
   time: \( O(\text{length of } H.\text{rootlist}) \)
   • Step 2 can potentially be linear in \( n \) (size of \( H \))

Decrease-Key (at node \( v \)):
1. If new key < parent key, cut sub-tree of node \( v \\
   time: \( O(1) \)
2. Cascading cuts up the tree as long as nodes are marked
   time: \( O(\text{number of consecutive marked nodes}) \)
   • Step 2 can potentially be linear in \( n \)

Exercises: Both operations can take \( \Theta(n) \) time in the worst case!
Cost of Delete-Min & Decrease-Key

• Cost of delete-min and decrease-key can be $\Theta(n)$...
  – Seems a large price to pay to get insert and merge in $O(1)$ time

• Maybe, the operations are efficient most of the time?
  – It seems to require a lot of operations to get a long rootlist and thus, an expensive consolidate operation
  – In each decrease-key operation, at most one node gets marked:
    We need a lot of decrease-key operations to get an expensive decrease-key operation

• Can we show that the average cost per operation is small?

• We can $\rightarrow$ requires amortized analysis
Amortization

- Consider sequence $o_1, o_2, \ldots, o_n$ of $n$ operations (typically performed on some data structure $D$)

- $t_i$: execution time of operation $o_i$
- $T := t_1 + t_2 + \ldots + t_n$: total execution time

- The execution time of a single operation might vary within a large range (e.g., $t_i \in [1, O(i)]$)

- The worst case overall execution time might still be small
  \( \rightarrow \) average execution time per operation might be small in the worst case, even if single operations can be expensive
Analysis of Algorithms

- Best case
- Worst case
- Average case
- Amortized worst case

What is the average cost of an operation in a worst case sequence of operations?
Example: Binary Counter

Incrementing a binary counter: determine the bit flip cost:

<table>
<thead>
<tr>
<th>Operation</th>
<th>Counter Value</th>
<th>Cost</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>00000</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>00001</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>00010</td>
<td>2</td>
</tr>
<tr>
<td>3</td>
<td>00011</td>
<td>1</td>
</tr>
<tr>
<td>4</td>
<td>00100</td>
<td>3</td>
</tr>
<tr>
<td>5</td>
<td>00101</td>
<td>1</td>
</tr>
<tr>
<td>6</td>
<td>00110</td>
<td>2</td>
</tr>
<tr>
<td>7</td>
<td>00111</td>
<td>1</td>
</tr>
<tr>
<td>8</td>
<td>01000</td>
<td>4</td>
</tr>
<tr>
<td>9</td>
<td>01001</td>
<td>1</td>
</tr>
<tr>
<td>10</td>
<td>01010</td>
<td>2</td>
</tr>
<tr>
<td>11</td>
<td>01011</td>
<td>1</td>
</tr>
<tr>
<td>12</td>
<td>01100</td>
<td>3</td>
</tr>
<tr>
<td>13</td>
<td>01101</td>
<td>1</td>
</tr>
</tbody>
</table>
Accounting Method

Observation:
• Each increment flips exactly one 0 into a 1

\[ 0010001111 \Rightarrow 0010010000 \]

Idea:
• Have a bank account (with initial amount 0)
• Paying $x$ to the bank account costs $x$
• Take “money” from account to pay for expensive operations

Applied to binary counter:
• Flip from 0 to 1: pay 1 to bank account (cost: 2)
• Flip from 1 to 0: take 1 from bank account (cost: 0)
• Amount on bank account = number of ones
  \[ \Rightarrow \text{We always have enough “money” to pay!} \]
### Accounting Method

<table>
<thead>
<tr>
<th>Op.</th>
<th>Counter</th>
<th>Cost</th>
<th>To Bank</th>
<th>From Bank</th>
<th>Net Cost</th>
<th>Credit</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>00000</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>00001</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>00010</td>
<td>2</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>00011</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>00100</td>
<td>3</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>00101</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>00110</td>
<td>2</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>00111</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>01000</td>
<td>4</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>9</td>
<td>01001</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>10</td>
<td>01010</td>
<td>2</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Potential Function Method

• Most **generic** and **elegant** way to do amortized analysis!
  – But, also more abstract than the others...

• State of data structure / system: $S \in S$ (state space)

  Potential function $\Phi: S \rightarrow \mathbb{R}_{\geq 0}$

• Operation $i$:
  – $t_i$: actual cost of operation $i$
  – $S_i$: state after execution of operation $i$ ($S_0$: initial state)
  – $\Phi_i := \Phi(S_i)$: potential after exec. of operation $i$
  – $a_i$: amortized cost of operation $i$:

$$a_i := t_i + \Phi_i - \Phi_{i-1}$$
Potential Function Method

Operation $i$:

- actual cost: $t_i$
- amortized cost: $a_i = t_i + \Phi_i - \Phi_{i-1}$

Overall cost:

$$T := \sum_{i=1}^{n} t_i = \left( \sum_{i}^{n} a_i \right) + \Phi_0 - \Phi_n$$
Binary Counter: Potential Method

• Potential function:
  \( \Phi \): number of ones in current counter

• Clearly, \( \Phi_0 = 0 \) and \( \Phi_i \geq 0 \) for all \( i \geq 0 \)

• Actual cost \( t_i \):
  - 1 flip from 0 to 1
  - \( t_i - 1 \) flips from 1 to 0

• Potential difference: \( \Phi_i - \Phi_{i-1} = 1 - (t_i - 1) = 2 - t_i \)

• Amortized cost: \( a_i = t_i + \Phi_i - \Phi_{i-1} = 2 \)
Back to Fibonacci Heaps

• Worst-case cost of a single delete-min or decrease-key operation is $\Omega(n)$

• Can we prove a small worst-case amortized cost for delete-min and decrease-key operations?

Remark:
• Data structure that allows operations $O_1, \ldots, O_k$

• We say that operation $O_p$ has amortized cost $a_p$ if for every execution the total time is

$$T \leq \sum_{p=1}^{k} n_p \cdot a_p,$$

where $n_p$ is the number of operations of type $O_p$
Amortized Cost of Fibonacci Heaps

• Initialize-heap, is-empty, get-min, insert, and merge have worst-case cost $O(1)$

• Delete-min has amortized cost $O(\log n)$
• Decrease-key has amortized cost $O(1)$

• Starting with an empty heap, any sequence of $n$ operations with at most $n_d$ delete-min operations has total cost (time) 

\[ T = O(n + n_d \log n). \]

• Cost for Dijkstra: $O(|E| + |V| \log |V|)$