CS711008Z Algorithm Design and Analysis

Lecture 2. Analysis techniques [1]

Dongbo Bu

Institute of Computing Technology
Chinese Academy of Sciences, Beijing, China

[1] The slides are made based on Ch. 17 of Introduction to Algorithms, and Ch. 2 of Algorithm Design. Some slides are excerpted from Kevin Wayne's slides with permission.

What is efficiency?

- **Definition 1:** An algorithm is efficient if, when implemented, it runs quickly on real input instances.
- **Questions:**
  - What is the platform?
  - Is the algorithm implemented well?
  - What is a “real” instance?
  - How well, or badly, does the algorithm scale with the instance size?
  - Both $Algo1$ and $Algo2$ perform well for a small instance; however, on a larger instance, one algorithm may be still fast, while the other one are very slow;

What is efficiency? cont'd

- **Definition 2:** An algorithm is efficient if it achieves qualitatively better worst-case performance, at an analytical level, than brute-force search.
- **Questions:**
  - Good: Algorithms better than brute-force search nearly always contains a valuable idea to make it work, and tell us the something about the intrinsic structure.
  - Bad: "quantatively" requires the actual running time of algorithm; thus, we should derive the running time carefully.

What is efficiency? cont'd

- **Definition 3:** An algorithm is efficient if it has a polynomial worst-case running time (known as Cobham-Edmonds thesis)
- **Justification:** It really works in practice.
  - In practice, the polynomial time algorithm that people develop almost always have low constant and low exponents;
  - Breaking the exponential barrier of brute-force usually means the exposition of problem structure.
- **Exceptions:**
  - Some polynomial-time algorithms have a high constant or high exponents, thus unpractical.
  - Some exponential-time algorithms work well in practice since the worst-case is rare.

Algorithm analysis

1. **Worst-case analysis:** the largest possible time on a problem instance with size n;
2. **Average-case analysis:** analyse average running time over all inputs with a known distribution;
3. **Amortized analysis:** worst case bound on a sequence of operations;

Note: Running time is usually measured in terms of elementary operations, say **comparison** in sort algorithm. Intuitively, an elementary operation takes 1 unit time, and the running time is measured using the number of elementary operations.

Average-case analysis

Average-case analysis

- Objective: analyze average running time over a distribution of inputs
- Example: QUICKSORT
  1. Worst-case complexity: $O(n^2)$
  2. Average-case complexity: $O(n \log n)$ if input is uniformly random

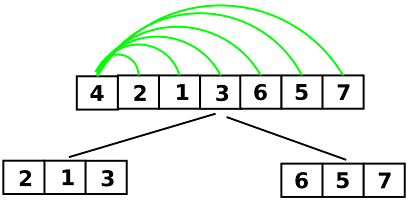
An example

Input: an array $A[1..n]$ of numbers
Output: sorted array
QUICKSORT algorithm

1: Pick an element, say the first element, from A. This element is called a pivot;
2: Partition A into two sub-lists, one consisting of elements less than the pivot, and another one consisting of elements larger than the pivot;
3: Recursively sort the sub-list of lesser elements and the sub-list of greater elements.

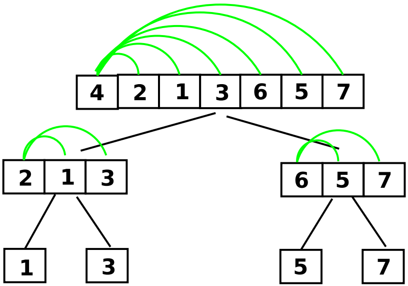
Best-case

- The most balanced case: partitioning A into two sub-lists of size $\frac{n}{2}$.



Best-case

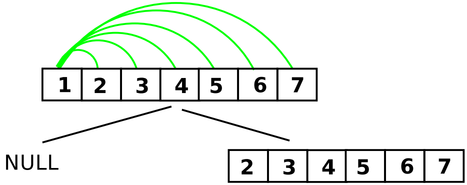
- The most balanced case: partitioning A into two sub-lists of size $\frac{n}{2}$.



Time: $T(n) = O(n) + 2T(\frac{n}{2}) = O(n \log_2 n)$

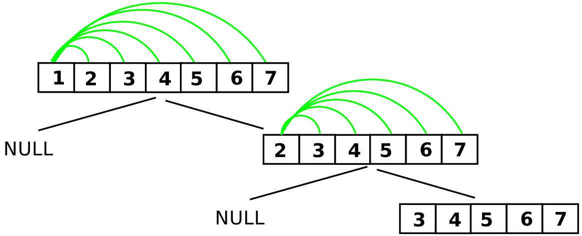
Worst-case

- The most unbalanced case: partitioning A into two sub-lists with size 1 and $n-1$.



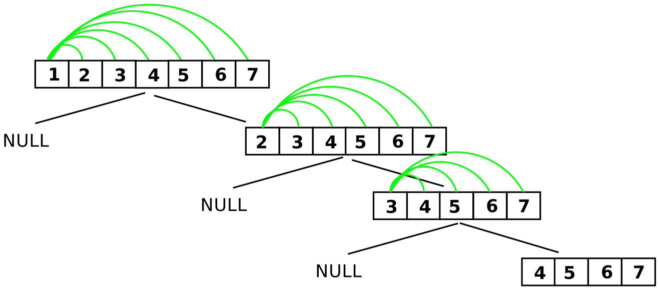
Worst-case

- The most unbalanced case: partitioning A into two sub-lists with size 1 and $n-1$.

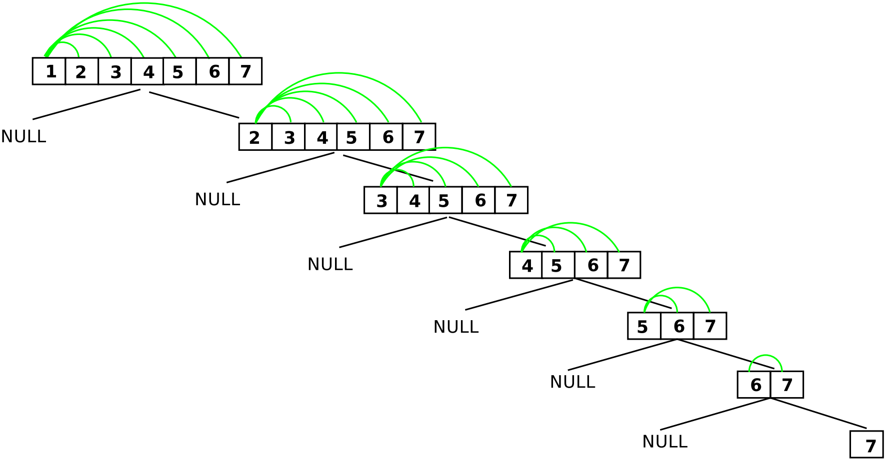


Worst-case

- The most unbalanced case: partitioning A into two sub-lists with size 1 and $n-1$.



Worst-case



Time: $T(n) = O(n) + T(n-1) = O(n^2)$

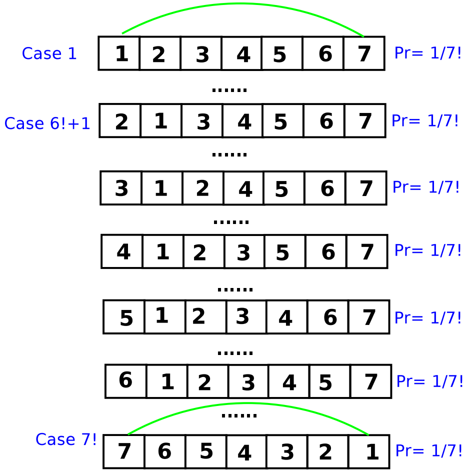
Average-case

- Assumption: the input is a random permutation

Case 1	1	2	3	4	5	6	7	Pr= 1/7!	T(n)=0(n^2)
				......					
Case 6!+1	2	1	3	4	5	6	7	Pr= 1/7!	.....
				......					
	3	1	2	4	5	6	7	Pr= 1/7!	.....
				......					
	4	1	2	3	5	6	7	Pr= 1/7!	T(n)=0(n log n)
				......					
	5	1	2	3	4	6	7	Pr= 1/7!	.....
				......					
	6	1	2	3	4	5	7	Pr= 1/7!	.....
				......					
Case 7!	7	6	5	4	3	2	1	Pr= 1/7!	T(n)=0(n^2)

- Objective: what is the average cost?

Average-case



- Note that $\Pr(1 \text{ compared with } 7) = \frac{2}{7}$. Why?
- In general, we have $\Pr(\text{ i compared with j }) = \frac{2}{j-i+1}$

Consider every pair

$$
\begin{aligned}
E(\#Comparison) &= \sum_{i=1}^{n-1} \sum_{j=i+1}^{n} \frac{2}{j-i+1} && (1)\\
&= \sum_{i=1}^{n-1} \sum_{k=1}^{n-i} \frac{2}{k+1} && (2)\\
&< \sum_{i=1}^{n-1} \sum_{k=1}^{n} \frac{2}{k} && (3)\\
&\approx 2n \ln n && (4)\\
&\approx 1.39 n \log_2 n && (5)
\end{aligned}
$$

Note:

- Equation (2) comes from introducing an auxiliary variable $k = j - i$.
- This means that, on average, QUICKSORT performs only about 39% worse than in its best case.

Amortized analysis

Amortized analysis

- Motivation: given a **sequence** of operations, the vast majority of the operations are cheap, but some rare operations within the sequence might be expensive; thus a standard worst-case analysis might be overly pessimistic.
- Objective: to give a tighter bound for a **sequence** of operations.
- Basic idea: when the expensive operations are particularly rare, their costs can be "spread out" (amortized) to all operations. If the artificial amortized costs are still cheap, we will have a tighter bound of the whole sequence of operations.
- Example: serving coffee in a bar

Amortized analysis versus average-case analysis

Amortized analysis differs from average-case analysis in:

- Average-case analysis: **average over all input**, e.g., QuickSort algorithm performs well on "average" over all possible input even if it performs very badly on certain input.
- Amortized analysis: **average over operations**, e.g., TableInsertion algorithm performs well on "average" over all operations even if some operations use a lot of time.

Stack with MULTIPOP operation

Problem: A Stack with MultiPop operation

Input: an array $A[1..n]$, an integer K;

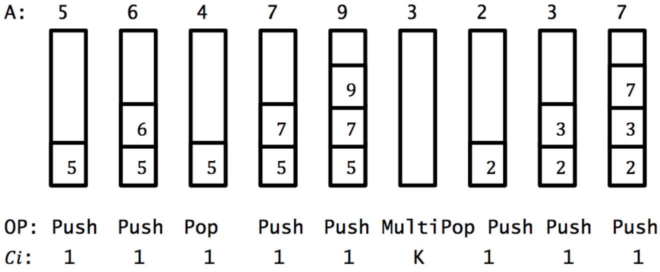
A sequence of n operations:

for $i = 1$ to n **do**
  if $A[i] \geq A[i-1]$ **then**
    Push($A[i]$);
  else if $A[i] \leq A[i-1] - K$ **then**
    MultiPop(S, K);
  else
    Pop();
  end if
end for

MultiPop(S, K)

while S is not empty and $k > 0$ **do**
  Pop(S);
  $k - -$;
end while

An example



Objective

For each operation assign an **amortized cost** $\widehat{C_i}$ to bound the actual total cost.

In other words, we need to show that for **any sequence of n operations**, we have $T(n) = \sum_{i=1}^{n} C_i \leq \sum_{i=1}^{n} \widehat{C_i}$. Here, C_i denotes the **actual cost** of step i.

Cursory analysis versus tighter analysis

- In a sequence of operations, some operations may be cheap, but some operations may be expensive, say $\textsc{MultiPop}()$.
- Cursory analysis: $\textsc{MultiPop}()$ step may take $O(n)$ time; thus, $T(n) = \sum_{i=1}^{n} C_i \leq n^2$
- However, the worst operation does not occur often.
- Therefore, the traditional worst-case **individual operation** analysis can give overly pessimistic bound.

Tighter analysis 1: aggregate technique

Tighter analysis 1: Aggregate technique

- Basic idea: all operations have the same AMORTIZED COST $\frac{1}{n}\sum_{i=1}^{n}\widehat{C_i}$
- Key observation: $\#Pop \leq \#Push$
- Thus, we have:

$$
\begin{aligned}
T(n) &= \sum_{i=1}^{n} C_i && (6)\\
&= \#Push + \#Pop && (7)\\
&\leq 2 \times \#Push && (8)\\
&\leq 2n && (9)
\end{aligned}
$$

- On average, the $MultiPop(K)$ step takes only $O(1)$ time rather than $O(K)$ time.

Tighter analysis 2: accounting technique

Tighter analysis 2: Accounting technique

- Basic idea: for each operation OP with actual cost C_{OP}, an amortized cost $\widehat{C_{OP}}$ is assigned such that for **any sequence of n operations**, $T(n) = \sum_{i=1}^{n} C_i \leq \sum_{i=1}^{n} \widehat{C_i}$.
- Intuition: If $\widehat{C_{op}} > C_{op}$, the overcharge will be stored as **prepaid credit**; the credit will be used later for the operations with $\widehat{C_{op}} < C_{op}$. The requirement that $\sum_{i=1}^{n} C_i \leq \sum_{i=1}^{n} \widehat{C_i}$ is essentially **credit never goes negative.**
- Example:

OP	Real Cost C_{op}	Amortized Cost $\widehat{C_{op}}$
PUSH	1	2
POP	1	0
MULTIPOP	k	0

- Credit: the number of items in the stack.

- Example:

OP	Real Cost C_{op}	Amortized Cost $\widehat{C_{op}}$
PUSH	1	2
POP	1	0
MULTIPOP	k	0

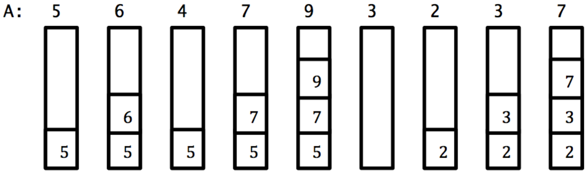
- In summary, starting from an empty stack, **any** sequence of n_1 PUSH, n_2 POP, and n_3 MULTIPOP operations takes at most $T(n) = \sum_{i=1}^{n} C_i \leq \sum_{i=1}^{n} \widehat{C_i} = 2n_1$. Here $n = n_1 + n_2 + n_3$.
- Note: when there are more than one type of operations, each type of operation might be assigned with different amortized cost.

Accounting method: "banker's view"

- Suppose you are renting a **"coin-operation"** machine, and are charged according to the number of operations.
- Two payment strategies:
  1. Pay actual cost for each operation: say pay $\$1$ for $\textsc{Push}$, $\$1$ for $\textsc{Pop}$, and $\$k$ for $\textsc{MultiPop}(\textsc{k})$.
  2. Open an account, and pay "average" cost for each operation: say pay $\$2$ for $\textsc{Push}$, $\$0$ for $\textsc{Pop}$, and $\$0$ for $\textsc{MultiPop}(\textsc{k})$.

  - If "average" cost $>$ actual cost: the extra will be deposited as *credit*.
  - If "average" cost $<$ actual cost: credit will be used to pay the actual cost.
- Constraint: $\sum_{i=1}^{n} C_i \leq \sum_{i=1}^{n} \widehat{C_i}$ for arbitrary n operations, i.e. you have enough **credit** in your account.

Accounting method: Intuition cont'd



OP:	Push	Push	Pop	Push	Push	MultiPop	Push	Push	Push
Ci:	1	1	1	1	1	K	1	1	1
$\widehat{Ci}$:	2	2	0	0	2	0	2	2	2
CREDIT:	1	2	1	2	3	0	1	2	3

- Credit: the number of items in the stack.
- Constraint: $\sum_{i=1}^{n} C_i \leq \sum_{i=1}^{n} \widehat{C_i}$ for arbitrary n operations, i.e. you have enough **credit** in your account.

Tighter analysis 3: potential function technique

Tighter analysis 3: Potential technique—"physicisit's view"

- Basic idea: sometimes it is not easy to set $\widehat{C_{op}}$ for each operation OP directly.
- Using potential function as a bridge, i.e. we assign a value to state rather than operation, and amortized costs are then calculated based on potential function.
- Potential function: $\Phi(S) : S \to R$. Here state S_i refers to the STATE of the stack after the i-th operation.
- Amortized cost setting: $\widehat{C_i} = C_i + \Phi(S_i) - \Phi(S_{i-1})$,
- Thus,

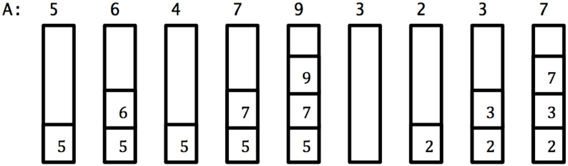
$$\sum_{i=1}^{n} \widehat{C_i} = \sum_{i=1}^{n} (C_i + \Phi(S_i) - \Phi(S_{i-1})) \quad (10)$$

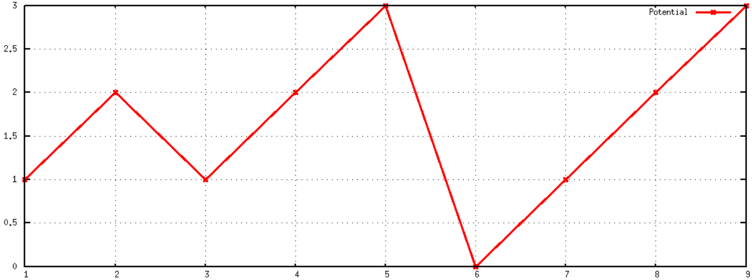
$$= \sum_{i=1}^{n} C_i + \Phi(S_n) - \Phi(S_0) \quad (11)$$

- Requirement: To guarantee $\sum_{i=1}^{n} C_i \leq \sum_{i=1}^{n} \widehat{C_i}$, it suffices to assure $\Phi(S_n) \geq \Phi(S_0)$.

Stack example: Potential changes

- **Definition:** $\Phi(S)$ denotes the number of items in stack. In fact, we simply **use "credit" as potential.**
- Correctness: $\Phi(S_i) \geq 0 = \Phi(S_0)$ for any i;





Potential function technique: amortized cost setting

Definition: $\Phi(S)$ denotes the number of items in stack;

- $\textsc{Push}$: $\Phi(S_i) - \Phi(S_{i-1}) = 1$

$$\widehat{C_i} = C_i + \Phi(S_i) - \Phi(S_{i-1}) \quad (12)$$
$$= 2 \quad (13)$$

- $\textsc{Pop}$: $\Phi(S_i) - \Phi(S_{i-1}) = -1$

$$\widehat{C_i} = C_i + \Phi(S_i) - \Phi(S_{i-1}) \quad (14)$$
$$= 0 \quad (15)$$

- $\textsc{MultiPop}$: $\Phi(S_i) - \Phi(S_{i-1}) = -\#Pop$

$$\widehat{C_i} = C_i + \Phi(S_i) - \Phi(S_{i-1}) \quad (16)$$
$$= 0 \quad (17)$$

- Thus, starting from an empty stack, **any sequence** of n_1 $\textsc{Push}$, n_2 $\textsc{Pop}$, and n_3 $\textsc{MultiPop}$ operations takes at most $T(n) = \sum_{i=1}^{n} C_i \leq \sum_{i=1}^{n} \widehat{C_i} = 2n_1$. Here $n = n_1 + n_2 + n_3$.

BinaryCounter problem

BinaryCounter problem: incrementing a binary counter

A sequence of n operations:

for $i = 1$ to n **do**
  Increment(A);
end for

Increment(A)

$i = 0$;
while $i \leq A.size()$ AND $A[i] == 1$ **do**
  $A[i] = 0$;
  $i++$;
end while
if $i \leq A.size()$ **then**
  $A[i] = 1$;
end if

Question: $T(n) \leq ?$

BinaryCounter operations: cursory analysis

Counter Value	A[7]	A[6]	A[5]	A[4]	A[3]	A[2]	A[1]	A[0]	Cost	Total Cost
0	0	0	0	0	0	0	0	0	0	0
1	0	0	0	0	0	0	0	1	1	1
2	0	0	0	0	0	0	1	0	2	3
3	0	0	0	0	0	0	1	1	1	4
4	0	0	0	0	0	1	0	0	3	7
5	0	0	0	0	0	1	0	1	1	8
6	0	0	0	0	0	1	1	0	2	10
7	0	0	0	0	0	1	1	1	1	11
8	0	0	0	0	1	0	0	1	4	15

- Cursory analysis: $T(n) \leq kn$ since an increment step might change all k bits.

Tighter analysis 1: aggregate technique

Tighter analysis 1: Aggregate technique

- Basic operations: `flip(1→0)`, `flip(0→1)`

$$
\begin{aligned}
T(n) &= \sum_{i=1}^{n} C_i \\
&= 1+2+1+3+1+2+1+4+... \\
&= \#flip\_at\_A0 + \#flip\_at\_A1 + ... + \#flip\_at\_Ak \\
&= n + \frac{n}{2} + \frac{n}{4} + ... \\
&\leq 2n
\end{aligned}
$$

- Amortized cost of each operation: $O(n)/n = O(1)$.

Tighter analysis 2: accounting technique

Tighter analysis 2: Accounting technique

Set amortized cost as follows:

OP	Real Cost C_{OP}	Amortized Cost $\widehat{C_{OP}}$
`flip(0→1)`	1	2
`flip(1→0)`	1	0

Key observation: $\#flip(0 \to 1) \geq \#flip(1 \to 0)$

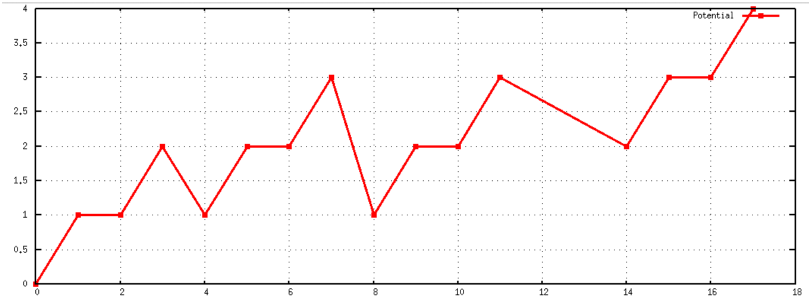
$$\begin{aligned} T(n) &= \sum_{i=1}^{n} C_i & (18) \\ &= \#flip(0 \to 1) + \#flip(1 \to 0) & (19) \\ &\leq 2\#flip(0 \to 1) & (20) \\ &\leq 2n & (21) \end{aligned}$$

Tighter analysis 3: potential function technique

Tighter analysis 3: Potential function technique

Definition: Set potential function as $\Phi(S) = \#1$ in counter

Counter Value	A[7]	A[6]	A[5]	A[4]	A[3]	A[2]	A[1]	A[0]	Cost	Total Cost
0	0	0	0	0	0	0	0	0	0	0
1	0	0	0	0	0	0	0	1	1	1
2	0	0	0	0	0	0	1	0	2	3
3	0	0	0	0	0	0	1	1	1	4
4	0	0	0	0	0	1	0	0	3	7
5	0	0	0	0	0	1	0	1	1	8
6	0	0	0	0	0	1	1	0	2	10
7	0	0	0	0	0	1	1	1	1	11
8	0	0	0	0	1	0	0	1	4	15



Tighter analysis: Potential technique cont'd

- **Definition:** Set potential function as $\Phi(S) = \#1$ in counter;
- At step i, the number of flips C_i is:

$$\begin{aligned} C_i &= \#flip^{(i)}_{0\to1} + \#flip^{(i)}_{1\to0} = 1 + \#flip^{(i)}_{1\to0} \quad (why?) \\ \Phi(S_i) &= \Phi(S_{i-1}) + 1 - \#flip^{(i)}_{1\to0} \\ \widehat{C_i} &= C_i + \Phi(S_i) - \Phi(S_{i-1}) \\ &\leq 2 \end{aligned}$$

- Thus we have

$$\begin{aligned} T(n) &= \sum_{i=1}^{n} C_i \\ &\leq \sum_{i=1}^{n} \widehat{C_i} \\ &\leq 2n \end{aligned}$$

- In other words, starting from $00....0$, a sequence of n INCREMENT operations takes at most $2n$ time.

DynamicTable problem

A practical problem

Practical problem:

- Suppose you are asked to develop a C++ compiler.
- `vector` is one of a C++ class templates to hold a set of objects. It supports the following operations:
  - `push_back`: to add a new object onto the tail;
  - `pop_back`: to pop out the last object;
- Recall that `vector` uses a **contiguous memory area** to store objects.
- Question: How to design an efficient **memory-allocation strategy** for `vector`?

DynamicTable problem

- In many applications, we do not know in advance how many objects will be stored in a table.
- Thus we have to allocate space for a table, only to find out later that it is not enough.
- Dynamic Expansion: When inserting a new item into a full table, the table must be reallocated with a larger size, and the objects in the original table must be copied into the new table.
- Dynamic Contraction: Similarly, if many objects have been removed from a table, it is worthwhile to reallocate the table with a smaller size.
- We will show a **memory allocation strategy** such that the amortized cost of insertion and deletion is $O(1)$, even if the actual cost of an operation is large when it triggers an expansion or contraction.

DynamicTable supporting TableInsertion operation only

Double-size strategy

$\textsc{Table\_Insert}(T, i)$

```
if size[T] == 0 then
    allocate a table with 1 slot;
    size[T] = 1;
end if
if num[T] == size[T] then
    allocate a new table with 2 × size[T] slots; //double size
    size[T] = 2 × size[T];
    copy all items into the new table;
    free the original table;
end if
insert the new item i into T;
num[T] + +;
```

`num[T]: #used slots`

`size[T]: total number of slots`

Consider a sequence of operations starting with an empty table:

```
Table T;
for i = 1 to n do
    Table_Insert(T, i);
end for
```

1. Insert(1)

C1: 1

TableInsert(2)

1. Insert(1)
2. Insert(2)

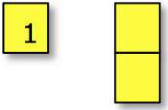
1

overflow

C1: 1

TableInsert(2)

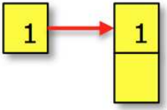
1. Insert(1)
2. Insert(2)



C1: 1

TableInsert(2)

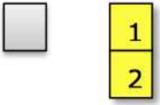
1. Insert(1)
2. Insert(2)



C1: 1

TableInsert(2)

1. Insert(1)
2. Insert(2)



C1: 1
C2: 2

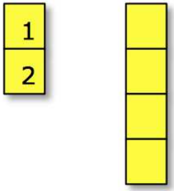
TableInsert(3)

1. Insert(1)
2. Insert(2)
3. Insert(3)

C1: 1
C2: 2

overflow

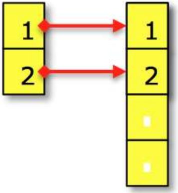
1. Insert(1)
2. Insert(2)
3. Insert(3)



C1: 1
C2: 2

TableInsert(3)

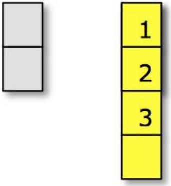
1. Insert(1)
2. Insert(2)
3. Insert(3)



C1: 1
C2: 2

TableInsert(3)

1. Insert(1)
2. Insert(2)
3. Insert(3)



C1: 1
C2: 2
C3: 3

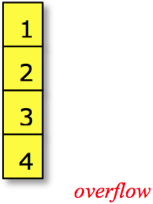
TableInsert(4)

1. Insert(1)
2. Insert(2)
3. Insert(3)
4. Insert(4)

C1: 1
C2: 2
C3: 3
C4: 1

TableInsert(5)

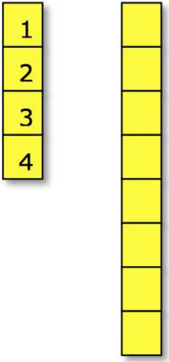
1. Insert(1)
2. Insert(2)
3. Insert(3)
4. Insert(4)
5. Insert(5)



C1: 1
C2: 2
C3: 3
C4: 1

TableInsert(5)

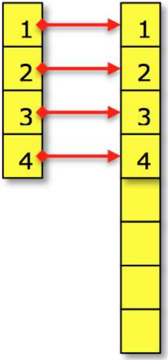
1. `Insert(1)`
2. `Insert(2)`
3. `Insert(3)`
4. `Insert(4)`
5. `Insert(5)`




```
C1: 1
C2: 2
C3: 3
C4: 1
```

TableInsert(5)

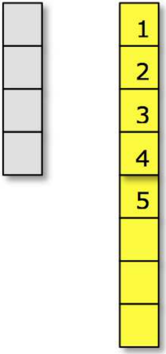
1. Insert(1)
2. Insert(2)
3. Insert(3)
4. Insert(4)
5. Insert(5)



C1: 1
C2: 2
C3: 3
C4: 1

TableInsert(5)

1. Insert(1)
2. Insert(2)
3. Insert(3)
4. Insert(4)
5. Insert(5)



C1: 1
C2: 2
C3: 3
C4: 1
C5: 5

Cursory analysis: $O(n^2)$

- Consider a sequence of operations starting with an empty table:

```
1: Table T;
2: for i = 1 to n do
3:     TABLE_INSERT(T, i);
4: end for
```

- What is the actual cost C_i of the ith operation? [2]

$$C_i = \begin{cases} i & \text{if } i-1 \text{ is an exact power of 2} \\ 1 & \text{otherwise} \end{cases}$$

- Here $C_i = i$ when the table is full, since we need to perform 1 insertion, and copy $i-1$ items into the new table.
- If n operations are performed, the worst-case cost of an operation will be $O(n)$.
- Thus, the total running time for a total of n operations is $O(n^2)$. **Not tight!**

[2] Here the cost is measured in terms of elementary insertions or deletions.

Tighter analysis 1: Aggregate technique

Aggregate method: **table expansions** are rare

- The $O(n^2)$ bound is not tight since **table expansion** doesn't occur often in the course of n operations.
- Specifically, **table expansion** occurs at the ith operation, where $i-1$ is an exact power of 2.

$$C_i = \begin{cases} i & \text{if } i-1 \text{ is an exact power of 2} \\ 1 & \text{otherwise} \end{cases}$$

i	1	2	3	4	5	6	7	8	9	10
$Size_i$	1	2	4	4	8	8	8	8	16	16
C_i	1	2	3	1	5	1	1	1	9	1

Aggregate method: rewriting C_i

- The $O(n^2)$ bound is not tight since **table expansion** doesn't occur often in the course of n operations.
- Specifically, **table expansion** occurs at the ith operation, where $i-1$ is an exact power of 2.

$$C_i = \begin{cases} i & \text{if } i-1 \text{ is an exact power of 2} \\ 1 & \text{otherwise} \end{cases}$$

- We decompose C_i as follows:

i	1	2	3	4	5	6	7	8	9	10
$Size_i$	1	2	4	4	8	8	8	8	16	16
C_i	1	1	1	1	1	1	1	1	1	1
		1	2		4				8	

Total cost of n operations

- The total cost of n operations is:

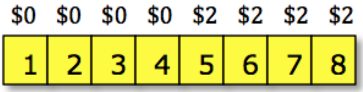
$$\begin{aligned}\sum_{i=1}^{n} C_i &= 1+2+3+1+5+1+1+1+9+1+\ldots \\ &= n + \sum_{j=0}^{\lfloor \lg n \rfloor} 2^j \\ &< n + 2n \\ &= 3n\end{aligned}$$

- Thus the amortized cost of an operation is 3.
- In other words, the average cost of each TableInsert operation is $O(n)/n = O(1)$.

Tighter analysis 2: Accounting technique

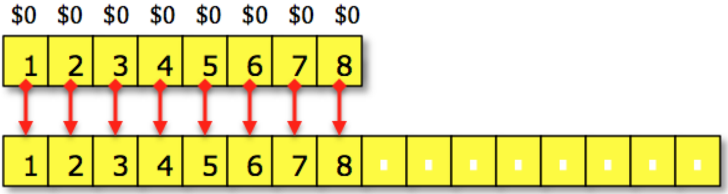
Tighter analysis 2: accounting technique

- For the i-th operation, an **amortized cost** $\widehat{C_i} = \$3$ is charged.
- This fee is consumed to perform subsequent operations.
- Any amount not immediately consumed is stored in a "bank" for use for subsequent operations.
- Thus for the i-th insertion, the $\$3$ is used as follows:
  - $\$1$ pays for the insertion **itself**;
  - $\$2$ is stored for **later table doubling**, including $\$1$ for copying one of the recent $\frac{i}{2}$ items, and $\$1$ for copying one of the old $\frac{i}{2}$ items.



Tighter analysis 2: accounting technique

- For the i-th operation, an **amortized cost** $\widehat{C_i} = \$3$ is charged.
- This fee is consumed to perform the operation.
- Any amount not immediately consumed is stored in a "bank" for use for subsequent operations.
- Thus for the i-th insertion, the $\$3$ is used as follows:
  - $\$1$ pays for the insertion **itself**;
  - $\$2$ is stored for **later table doubling**, including $\$1$ for copying one of the recent $\frac{i}{2}$ items, and $\$1$ for copying one of the old $\frac{i}{2}$ items.



Tighter analysis 2: accounting technique

- Key observation: the credit never goes negative. In other words, the sum of amortized cost provides an upper bound of the sum of actual costs.

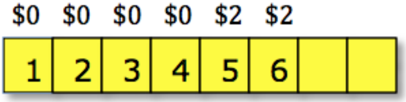
$$
\begin{aligned}
T(n) &= \sum_{i=1}^{n} C_i \\
&\leq \sum_{i=1}^{n} \widehat{C_i} \\
&= 3n
\end{aligned}
$$

i	1	2	3	4	5	6	7	8	9	10
$Size_i$	1	2	4	4	8	8	8	8	16	16
C_i	1	1	1	1	1	1	1	1	1	1
$\widehat{C_i}$	3	3	3	3	3	3	3	3	3	3
Credit	2	3	3	5	3	5	7	9	3	5

Tighter analysis 3: Potential function technique

Tighter analysis 3: potential function technique

- Motivation: sometimes it is not easy to find an appropriate amortized cost **directly**. An alternative way is to use a **potential function** as a bridge.
- Basic idea: the **bank account** can be viewed as potential function of the dynamic set. More specifically, we prefer a potential function $\Phi : \{T\} \rightarrow R$ with the following properties:
  - $\Phi(T) = 0$ immediately **after** an expansion;
  - $\Phi(T) = size[T]$ immediately **before** an expansion; thus, the next expansion can be paid for by the potential.
- A possibility: $\Phi(T) = 2 \times num[T] - size[T]$



$$\phi = 2num[T] - size[T] = 4$$

$\Phi(T) = 2 \times num[T] - size[T]$: an example

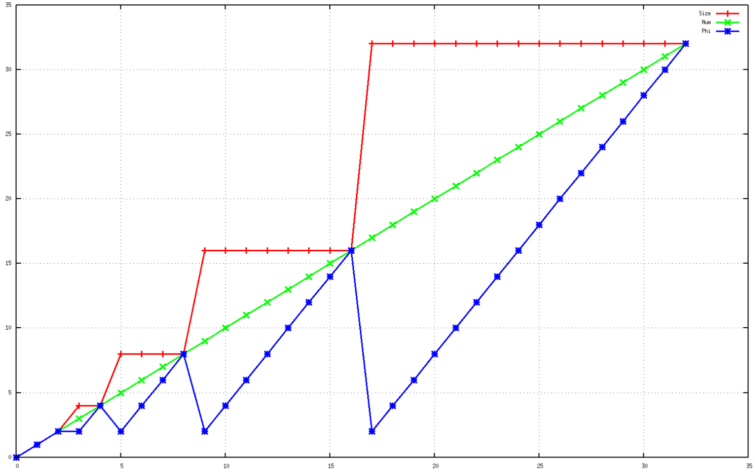


Figure: The effect of a sequence of n TableInsert on $size_i$ (red), num_i (green), and Φ_i (blue).

Correctness of $\Phi(T) = 2 \times num[T] - size[T]$

- Correctness: Initially $\Phi_0 = 0$, and it is easy to verify that $\Phi_i \geq \Phi_0$ since the table is always at least half full.
- The **amortized cost** $\widehat{C_i}$ with respect to Φ is defined as: $\widehat{C_i} = C_i + \Phi(T_i) - \Phi(T_{i-1})$.
- Thus $\sum_{i=1}^{n} \widehat{C_i} = \sum_{i=1}^{n} C_i + \Phi_n - \Phi_0$ is really an upper bound of the actual cost $\sum_{i=1}^{n} C_i$.

Calculate $\widehat{C_i}$ with respect to Φ

- Case 1: the i-th insertion does not trigger an expansion
- Then $size_i = size_{i-1}$. Here, num_i denotes the number of items after the i-th operations, $size_i$ denotes the table size, and T_i denotes the potential.

$$
\begin{aligned}
\widehat{C_i} &= C_i + \Phi_i - \Phi_{i-1} \\
&= 1 + (2num_i - size_i) - (2num_{i-1} - size_{i-1}) \\
&= 1 + 2 \\
&= 3
\end{aligned}
$$

```
1. Insert(1)
2. Insert(2)
3. Insert(3)
4. Insert(4)
```

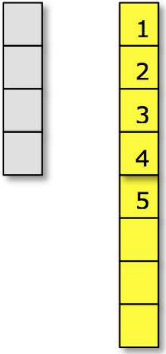

```
C1: 1
C2: 2
C3: 3
C4: 1
```

Calculate $\widehat{C_i}$ with respect to Φ

- Case 2: the i-th insertion triggers an expansion
- Then $size_i = 2 \times size_{i-1}$.

$$
\begin{aligned}
\widehat{C_i} &= C_i + \Phi_i - \Phi_{i-1} \\
&= num_i + (2num_i - size_i) - (2num_{i-1} - size_{i-1}) \\
&= num_i + 2 - (num_i - 1) \\
&= 3
\end{aligned}
$$

```
1. Insert(1)
2. Insert(2)
3. Insert(3)
4. Insert(4)
5. Insert(5)
```



C1: 1
C2: 2
C3: 3
C4: 1
C5: 5

Conclusion

Starting with an empty table, a sequence of n TableInsert operations cost $O(n)$ time in the worst case.

DynamicTable supporting TableInsert and TableDelete

TableDelete operation

- To implement TableDelete operation, it is simple to remove the specified item from the table, followed by a Contraction operation when the **load factor** (denoted as $\alpha(T) = \frac{num[T]}{size[T]}$) is small, so that the wasted space is not exorbitant.
- Specifically, when the number of the items in the table drops too low, we allocate a new, smaller space, copy the items from the old table to the new one, and finally free the original table.
- We would like the following two properties:
  1. The load factor is bounded below by a constant;
  2. The amortized cost of a table operation is bounded above by a constant.

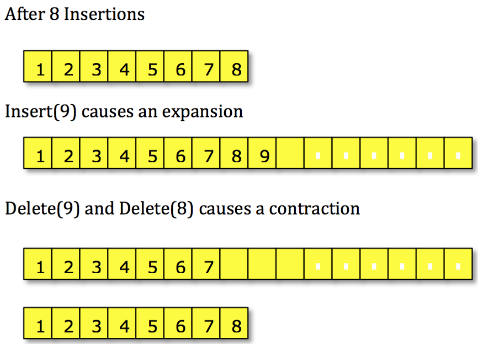
Trial 1: load factor $\alpha(T)$ never drops below 1/2

Trial 1: load factor $\alpha(T)$ never drops below 1/2

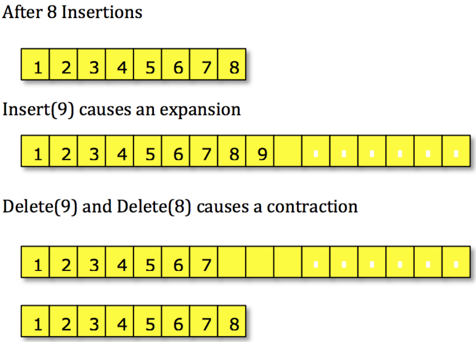
- A natural strategy is:
  - To double the table size when inserting an item into a full table;
  - To halve the table size when deletion causes $\alpha(T) < \frac{1}{2}$.
- The strategy guarantees that load factor $\alpha(T)$ never drops below 1/2.
- However, the amortized cost of an operation might be quite large.

An example of large amortized cost

- Consider a sequence of $n = 16$ operations:
  - The first 8 operations: `I, I, I,....`
  - The second 8 operations: `I, D, D, I, I, D, D, I, I,...`
- Note:
  - After the 8-th `I`, we have $num_{16} = size_{16} = 16$.
  - The 9-th `I` leads to a table expansion;
  - The following two `D` lead to a table contraction;
  - The following two `I` lead to a table expansion, and so on.



An example of large amortized cost



- The expansion/contraction takes $O(n)$ time, and there are n of them.
- Thus the total cost of n operations are $O(n^2)$, and the amortized cost of an operation is $O(n)$.

Trial 2: load factor $\alpha(T)$ never drops below $1/4$

Trial 1: load factor $\alpha(T)$ never drops below 1/2

- Another strategy is:
  - To double the table size when inserting an item into a full table;
  - To halve the table size when deletion causes $\alpha(T) < \frac{1}{4}$.
- The strategy guarantees that load factor $\alpha(T)$ never drops below 1/4.

Amortized analysis

- We start by defining a potential function $\Phi(T)$ that is 0 immediately after an expansion or contraction, and builds as $\alpha(T)$ increases to 1 or decreases to $\frac{1}{4}$.

$$\Phi(T) = \begin{cases} 2 \times num[T] - size[T] & \texttt{if } \alpha(T) \geq \frac{1}{2} \\ \frac{1}{2} size[T] - num[T] & \texttt{if } \alpha(T) \leq \frac{1}{2} \end{cases}$$

- Correctness: the potential is 0 for an empty table, and $\Phi(T)$ never goes negative. Thus, the total amortized cost of a sequence of n operations with respect to Φ is an upper bound of the actual cost.

Amortized cost of TableInsert operation

- Case 1: $\alpha_{i-1} \geq \frac{1}{2}$ and no expansion
- The amortized cost is:

$$
\begin{aligned}
\widehat{C_i} &= C_i + \Phi_i - \Phi_{i-1} \\
&= 1 + (2num_i - size_i) - (2num_{i-1} - size_{i-1}) \\
&= 1 + (2(num_{i-1} + 1) - size_i) - (2num_{i-1} - size_i) \\
&= 3
\end{aligned}
$$

```
1. Insert(1)
2. Insert(2)
3. Insert(3)
4. Insert(4)
```

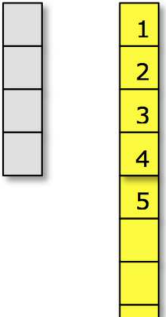

```
C1: 1
C2: 2
C3: 3
C4: 1
```

Amortized cost of TableInsert

- Case 2: $\alpha_{i-1} \geq \frac{1}{2}$ and an expansion was triggered
- The amortized cost is:

$$
\begin{aligned}
\widehat{C_i} &= C_i + \Phi_i - \Phi_{i-1} \\
&= num_i + (2num_i - size_i) - (2num_{i-1} - size_{i-1}) \\
&= num_{i-1} + 1 + (2(num_{i-1} + 1) - 2size_{i-1}) - (2num_{i-1} - \\
&= 3 + num_{i-1} - size_{i-1} \\
&= 3
\end{aligned}
$$

```
1. Insert(1)
2. Insert(2)
3. Insert(3)
4. Insert(4)
5. Insert(5)
```



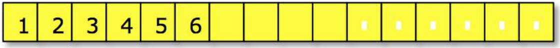

```
C1: 1
C2: 2
C3: 3
C4: 1
C5: 5
```

Amortized cost of TABLEINSERT

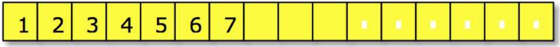
- Case 3: $\alpha_{i-1} < \frac{1}{2}$ and $\alpha_i < \frac{1}{2}$
- The amortized cost is:

$$
\begin{aligned}
\widehat{C_i} &= C_i + \Phi_i - \Phi_{i-1} \\
&= 1 + (\frac{1}{2} size_i - num_i) - (\frac{1}{2} size_{i-1} - num_{i-1}) \\
&= 1 + (\frac{1}{2} size_i - num_i) - (\frac{1}{2} size_i - (num_i - 1)) \\
&= 0
\end{aligned}
$$

num = 6, size = 16, phi = 2



num = 7, size=16, phi = 1



Amortized cost of TableInsert I

- Case 4: $\alpha_{i-1} < \frac{1}{2}$ but $\alpha_i \geq \frac{1}{2}$
- The amortized cost is:

$$
\begin{aligned}
\widehat{C_i} &= C_i + \Phi_i - \Phi_{i-1} \\
&= 1 + (2num_i - size_i) - (\frac{1}{2}size_{i-1} - num_{i-1}) \\
&= 1 + (2(num_{i-1} + 1) - size_{i-1}) - (\frac{1}{2}size_{i-1} - num_{i-1}) \\
&= 3num_{i-1} - \frac{3}{2}size_{i-1} + 3 \\
&= 3\alpha_{i-1}num_{i-1} - \frac{3}{2}size_{i-1} + 3 \\
&< \frac{3}{2}size_{i-1} - \frac{3}{2}size_{i-1} + 3 \\
&= 3
\end{aligned}
$$

Amortized cost of TABLEINSERT II

num = 7, size = 16, phi = 1

num = 8, size = 16, phi = 0

Amortized cost of TableDelete operation

Amortized cost of TABLEDELETE

- Case 1: $\alpha_{i-1} < \frac{1}{2}$ and no contraction
- The amortized cost is:

$$
\begin{aligned}
\widehat{C_i} &= C_i + \Phi_i - \Phi_{i-1} \\
&= 1 + (\frac{1}{2}size_i - num_i) - (\frac{1}{2}size_{i-1} - num_{i-1}) \\
&= 1 + (\frac{1}{2}size_{i-1} - (num_{i-1} - 1)) - (\frac{1}{2}size_{i-1} - num_{i-1}) \\
&= 2
\end{aligned}
$$

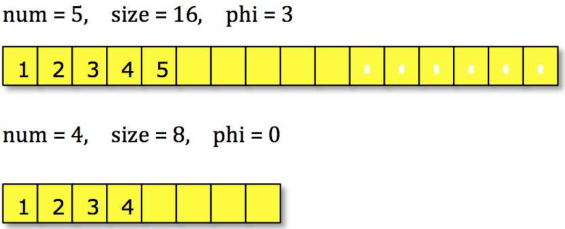
num = 7, size = 16, phi = 1

num = 6, size = 16, phi = 2

Amortized cost of TABLEDELETE

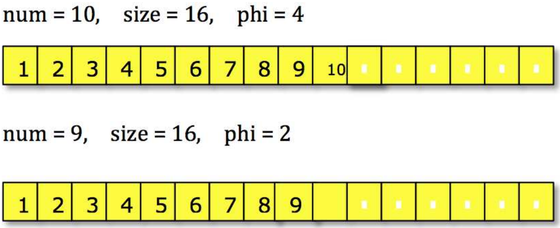
- Case 2: $\alpha_{i-1} < \frac{1}{2}$ and a contraction was triggered
- The amortized cost is:

$$
\begin{aligned}
\widehat{C_i} &= C_i + \Phi_i - \Phi_{i-1} \\
&= num_i + 1 + (\frac{1}{2}size_i - num_i) - (\frac{1}{2}size_{i-1} - num_{i-1}) \\
&= num_{i-1} + (\frac{1}{4}size_{i-1} - (num_{i-1} - 1)) - (\frac{1}{2}size_{i-1} - num \\
&= 1 + num_{i-1} - \frac{1}{4}size_{i-1} \\
&= 1
\end{aligned}
$$



- Case 3: $\alpha_{i-1} \geq \frac{1}{2}$ and $\alpha_i \geq \frac{1}{2}$
- The amortized cost is:

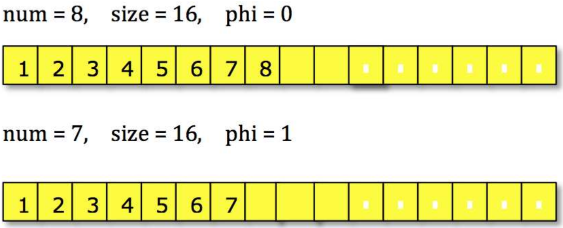
$$
\begin{aligned}
\widehat{C_i} &= C_i + \Phi_i - \Phi_{i-1} \\
&= 1 + (2num_i - size_i) - (2num_{i-1} - size_{i-1}) \\
&= 1 + (2(num_{i-1} + 1) - size_{i-1}) - (2num_{i-1} - size_{i-1}) \\
&= 3
\end{aligned}
$$



Amortized cost of TABLEINSERT

- Case 4: $\alpha_{i-1} \geq \frac{1}{2}$ and $\alpha_i < \frac{1}{2}$
- The amortized cost is:

$$\begin{aligned}
\widehat{C_i} &= C_i + \Phi_i - \Phi_{i-1} \\
&= 1 + (\frac{1}{2} size_i - num_i) - (2num_{i-1} - size_{i-1}) \\
&= 1 + (\frac{1}{2} size_{i-1} - (num_{i-1} - 1)) - (2num_{i-1} - size_{i-1}) \\
&= 2 + \frac{3}{2} size_{i-1} - 3num_{i-1} \\
&\leq 2
\end{aligned}$$



Conclusion

In summary, since the amortized cost of each operation is bounded above by a constant, the actual cost of any sequence of n TableInsert and TableDelete operations on a dynamic table is $O(n)$ if starting with an empty table.

More examples

We will talk about the following examples later:

- Binomial heap and Fibonacci heap
- Splay-tree
- Union-Find