Data Structures (II)

Lau Chi Yung

February 24, 2018
Content

1. Hash Table
2. Binary Heap
3. Binary Search Tree
**Problem Description**

$N$ operations:
- insert $X$
- remove $X$
- exists $X$

Constraints:
- $1 \leq N \leq 1000$
- $1 \leq X \leq 1000$

**Input**

```
7
insert 39
insert 12
insert 74
exists 39
exists 74
remove 12
exists 12
```

**Output**

```
yes
yes
no
```
Solution: plain array

<table>
<thead>
<tr>
<th>Inserted Value</th>
<th>Array Content</th>
</tr>
</thead>
<tbody>
<tr>
<td>insert 39</td>
<td>39</td>
</tr>
<tr>
<td>insert 12</td>
<td>39 12</td>
</tr>
<tr>
<td></td>
<td>...</td>
</tr>
<tr>
<td>insert 25</td>
<td>39 12 74 93 62 1 50 25</td>
</tr>
</tbody>
</table>
Solution: plain array

<table>
<thead>
<tr>
<th>Insert 39</th>
<th>39</th>
</tr>
</thead>
<tbody>
<tr>
<td>Insert 12</td>
<td>39</td>
</tr>
<tr>
<td>...</td>
<td></td>
</tr>
<tr>
<td>Insert 25</td>
<td>39</td>
</tr>
</tbody>
</table>

| Remove 12 | 39 | 74 | 93 | 62 | 1 | 50 | 25 |
| or        |    |    |    |    |   |    |    |
| or        | 39 | 25 | 74 | 93 | 62 | 1 | 50 |
| or        | ?  |    |    |    |    |    |    |
Solution: plain array

| insert 39 | 39 |
| insert 12 | 39 12 |
| insert 25 | 39 12 74 93 62 1 50 25 |

\[
\begin{array}{cccccccc}
39 & 74 & 93 & 62 & 1 & 50 & 25 \\
39 & 74 & 93 & 62 & 1 & 50 & 25 \\
39 & 25 & 74 & 93 & 62 & 1 & 50 \\
\end{array}
\]

or

\[
\begin{array}{cccccccc}
39 & 74 & 93 & 62 & 1 & 50 & 25 \\
\end{array}
\]

or

? 

exists 74 perform linear search
Solution: plain array

\( N \) operations:
- insert \( X \)
- remove \( X \)
- exists \( X \)

Constraints:
- \( 1 \leq N \leq 1000 \)
- \( 1 \leq X \leq 1000 \)

Time complexity:
- insert \( O(1) \)
- remove \( O(N) \)
- exists \( O(N) \)

For \( N \) operations, worst case \( O(N^2) \)
Solution: sorted array

<table>
<thead>
<tr>
<th>Operation</th>
<th>Array</th>
</tr>
</thead>
<tbody>
<tr>
<td>insert 39</td>
<td>39</td>
</tr>
<tr>
<td>insert 12</td>
<td>12 39</td>
</tr>
<tr>
<td></td>
<td>...</td>
</tr>
<tr>
<td>insert 25</td>
<td>1 12 25 39 50 62 74 93</td>
</tr>
</tbody>
</table>
Solution: sorted array

<table>
<thead>
<tr>
<th></th>
<th>1</th>
<th>25</th>
<th>39</th>
<th>50</th>
<th>62</th>
<th>74</th>
<th>93</th>
</tr>
</thead>
<tbody>
<tr>
<td>insert 25</td>
<td>1</td>
<td>12</td>
<td>25</td>
<td>39</td>
<td>50</td>
<td>62</td>
<td>74</td>
</tr>
<tr>
<td>insert 12</td>
<td>12</td>
<td>39</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>insert 39</td>
<td>39</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>remove 12</td>
<td>1</td>
<td>25</td>
<td>39</td>
<td>50</td>
<td>62</td>
<td>74</td>
<td>93</td>
</tr>
</tbody>
</table>
Solution: sorted array

| insert 39 | 39 |
| insert 12 | 12 39 |
| insert 25 | 1 12 25 39 50 62 74 93 |
| remove 12 | 1 25 39 50 62 74 93 |

exists 74  perform binary search
Solution: sorted array

\( N \) operations:
- insert \( X \)
- remove \( X \)
- exists \( X \)

Constraints:
- \( 1 \leq N \leq 1000 \)
- \( 1 \leq X \leq 1000 \)

Time complexity:
- insert \( O(N) \)
- remove \( O(N) \)
- exists \( O(\log N) \)

For \( N \) operations, worst case \( O(N^2) \)
Solution: hybrid

\( N \) operations:
- insert \( X \)
- remove \( X \)
- exists \( X \)

Constraints:
- \( 1 \leq N \leq 1000 \)
- \( 1 \leq X \leq 1000 \)

<table>
<thead>
<tr>
<th>Dominant operation</th>
<th>Solution array</th>
</tr>
</thead>
<tbody>
<tr>
<td>insert</td>
<td>plain or sorted?</td>
</tr>
<tr>
<td>exists</td>
<td>plain or sorted?</td>
</tr>
</tbody>
</table>
Solution: hybrid

\( N \) operations:

- insert \( X \)
- remove \( X \)
- exists \( X \)

Constraints:

- \( 1 \leq N \leq 1000 \)
- \( 1 \leq X \leq 1000 \)

<table>
<thead>
<tr>
<th>Dominant operation</th>
<th>Solution array</th>
</tr>
</thead>
<tbody>
<tr>
<td>insert</td>
<td>plain or sorted?</td>
</tr>
<tr>
<td>exists</td>
<td>plain or sorted?</td>
</tr>
</tbody>
</table>
Solution: hybrid

\( N \text{ operations:} \)
- insert \( X \)
- remove \( X \)
- exists \( X \)

Constraints:
- \( 1 \leq N \leq 1000 \)
- \( 1 \leq X \leq 1000 \)

<table>
<thead>
<tr>
<th>Dominant operation</th>
<th>Solution array</th>
</tr>
</thead>
<tbody>
<tr>
<td>insert</td>
<td>plain or sorted?</td>
</tr>
<tr>
<td>exists</td>
<td>plain or sorted?</td>
</tr>
</tbody>
</table>
Solution: hybrid

$N$ operations:

- insert $X$
- remove $X$
- exists $X$

Constraints:

- $1 \leq N \leq 1000$
- $1 \leq X \leq 1000$

<table>
<thead>
<tr>
<th>Dominant operation</th>
<th>Solution array</th>
</tr>
</thead>
<tbody>
<tr>
<td>insert</td>
<td>plain or sorted?</td>
</tr>
<tr>
<td>exists</td>
<td>plain or sorted?</td>
</tr>
</tbody>
</table>

For $N$ operations, still $O(N^2)$
Problem Description - Level Up

$L$ operations:
- insert $X$
- remove $X$
- exists $X$

Constraints:
- $1 \leq N \leq 10000$
- $1 \leq X \leq 1000$
Solution: counting array

Similar idea with counting sort

<table>
<thead>
<tr>
<th>Insert</th>
<th>Array</th>
</tr>
</thead>
<tbody>
<tr>
<td>12</td>
<td>![Array with insert 12]</td>
</tr>
<tr>
<td>17</td>
<td>![Array with insert 17]</td>
</tr>
<tr>
<td>6</td>
<td>![Array with insert 6]</td>
</tr>
</tbody>
</table>
Solution: counting array

Similar idea with counting sort

<table>
<thead>
<tr>
<th>Insert 12</th>
<th>1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17</th>
</tr>
</thead>
<tbody>
<tr>
<td>Insert 17</td>
<td>1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17</td>
</tr>
<tr>
<td>Insert 6</td>
<td>1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17</td>
</tr>
<tr>
<td>Remove 12</td>
<td>1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17</td>
</tr>
</tbody>
</table>
Solution: counting array

Similar idea with counting sort

insert 12

<table>
<thead>
<tr>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>10</th>
<th>11</th>
<th>12</th>
<th>13</th>
<th>14</th>
<th>15</th>
<th>16</th>
<th>17</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

insert 17

<table>
<thead>
<tr>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>10</th>
<th>11</th>
<th>12</th>
<th>13</th>
<th>14</th>
<th>15</th>
<th>16</th>
<th>17</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>1</td>
<td></td>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

insert 6

<table>
<thead>
<tr>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>10</th>
<th>11</th>
<th>12</th>
<th>13</th>
<th>14</th>
<th>15</th>
<th>16</th>
<th>17</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>1</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

remove 12

<table>
<thead>
<tr>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>10</th>
<th>11</th>
<th>12</th>
<th>13</th>
<th>14</th>
<th>15</th>
<th>16</th>
<th>17</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

exists 12 array lookup
Solution: counting array

\( N \) operations:
- insert \( X \)
- remove \( X \)
- exists \( X \)

Constraints:
- \( 1 \leq N \leq 10000 \)
- \( 1 \leq X \leq 1000 \)

Time complexity:
- insert \( O(1) \)
- remove \( O(1) \)
- exists \( O(1) \)

For \( N \) operations, worst case \( O(N) \)
Problem Description - Level Up

$N$ operations:

- insert $X$
- remove $X$
- exists $X$

Constraints:

- $1 \leq N \leq 10000$
- $1 \leq X \leq 10^8$
- Memory limit: 32MB
Problem Description - Level Up

$N$ operations:
- insert $X$
- remove $X$
- exists $X$

Constraints:
- $1 \leq N \leq 10000$
- $1 \leq X \leq 10^8$
- Memory limit: 32MB

Can we use counting array?

```c
int a[100000000 + 1];
```

$4 \times 10^8 = 400$MB  MLE
Problem Description - Level Up

*N* operations:
- insert *X*
- remove *X*
- exists *X*

Constraints:
- 1 ≤ *N* ≤ 10000
- 1 ≤ *X* ≤ 10^8
- Memory limit: 32MB

“*int*” contains 32 bits

```cpp
int a[3125000 + 1];
```

4 × 3125000 = 12.5MB

```cpp
insert *X*  a[*X*/32] |= 1<<*X*%32
remove *X*  a[*X*/32] &= ~(1<<*X*%32)
exists *X*  a[*X*/32] & 1<<*X*%32
```
Problem Description - Level Up Again

\( N \) operations:
- insert \( X \)
- remove \( X \)
- exists \( X \)

Constraints:
- \( 1 \leq N \leq 10000 \)
- \( 1 \leq X \leq 10^{18} \)
- Memory limit: 32MB
Problem Description - Level Up Again

\( N \) operations:

- insert \( X \)
- remove \( X \)
- exists \( X \)

Observation:
There can be at most 10000 distinct values of \( X \)

Constraints:

- \( 1 \leq N \leq 10000 \)
- \( 1 \leq X \leq 10^{18} \)
- Memory limit: 32MB
Problem Description - Level Up Again

\( N \) operations:

- insert \( X \)
- remove \( X \)
- exists \( X \)

Constraints:

- \( 1 \leq N \leq 10000 \)
- \( 1 \leq X \leq 10^{18} \)
- Memory limit: 32MB

Observation:

There can be at most 10000 distinct values of \( X \)

Perform discretization:

1. Preprocess all \( X \) into a sorted array
2. Using binary search, transform the range of \( X \) from \([1, 10^{18}]\) to \([1, 10000]\)
3. Then use counting array
Problem Description - Level Up Yet Again

$N$ operations:

- insert $X$
- remove $X$
- exists $X$

Constraints:

- $1 \leq N \leq 10000$
- $1 \leq X \leq 10^{18}$
- Memory limit: 32MB
- Preprocess of operations is not allowed (how?)
Problem Description - Level Up Yet Again

$N$ operations:
- insert $X$
- remove $X$
- exists $X$

Constraints:
- $1 \leq N \leq 10000$
- $1 \leq X \leq 10^{18}$
- Memory limit: 32MB
- Preprocess of operations is not allowed (how?)
- Perhaps the task is interactive
Problem Description - Level Up Yet Again

\( \text{\textit{N}} \) operations:
- insert \( X \)
- remove \( X \)
- exists \( X \)

Constraints:
- \( 1 \leq \text{\textit{N}} \leq 10000 \)
- \( 1 \leq X \leq 10^{18} \)
- Memory limit: 32MB
- Preprocess of operations is not allowed (how?)

- Perhaps the task is interactive
- Perhaps each operation’s input depends on the output of previous operations
We will encounter only 10000 integers in $[1, 10^{18}]$.

Intuitively, $X \mod 10^8$ will *probably* be distinct.

Assume for any $X_i \neq X_j$, $X_i \mod 10^8 \neq X_j \mod 10^8$.

We can replace all $X$ with $X \mod 10^8$ and use bitwise counting array!

How *likely* will our assumption hold?
Birthday problem

Probability that all 23 people have distinct birthdays

\[
P = \frac{365}{365} \times \frac{364}{365} \times \frac{363}{365} \times \cdots \times \frac{343}{365} \approx 0.492703
\]

For 70 people, the probability is 0.000840
Birthday problem

Probability that all 23 people have distinct birthdays

\[ P = \frac{365}{365} \times \frac{364}{365} \times \frac{363}{365} \times \cdots \times \frac{343}{365} \approx 0.492703 \]

For 70 people, the probability is 0.000840

Assume \( X \) is uniformly distributed over \([1, 10^{18}]\)

Then \( X \mod 10^8 \) is uniformly distributed over \([0, 10^8 - 1]\)

Probability that all 10000 \( X \mod 10^8 \) are distinct

\[ P = \frac{10^8}{10^8} \times \cdots \times \frac{10^8 - 9999}{10^8} \approx 0.606551 \]

Not too bad.
### Birthday problem

\[
P = \frac{10^8}{10^8} \times \cdots \times \frac{10^8 - 9999}{10^8} \approx 0.606551
\]

Had we used \(2 \times 10^8\) (which amounts to 25MB in bitwise counting array),

\[
P = \frac{2 \times 10^8}{2 \times 10^8} \times \cdots \times \frac{2 \times 10^8 - 9999}{2 \times 10^8} \approx 0.778817
\]

Intuitively, with more buckets available, less likely will collision occur.

This does not work well for batched testcases.

Probablility that we get all 5 testcases correct in a batch
\[= 0.778817^5 = 0.286534\]
Collision Handling 1

Rather than giving up on collisions, we try to handle them.

Want to map $X$ into $X \mod 10$.

**insert $X$:** put $X$ in the *nearest* empty bucket after $X \mod 10$

<table>
<thead>
<tr>
<th>insert 12</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td>12</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>insert 34</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>insert 22</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>insert 42</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
</tbody>
</table>
Collision Handling 1

remove $X$: search for the bucket containing $X$, replace it with $-1$.

\[
\begin{array}{ccccccc}
0 & 1 & 2 & 3 & 4 & 5 & 6 \\
\hline
12 & 22 & 34 & 42 & & & \\
\end{array}
\]

remove 22

\[
\begin{array}{ccccccc}
0 & 1 & 2 & 3 & 4 & 5 & 6 \\
\hline
12 & -1 & 34 & 42 & & & \\
\end{array}
\]

exists $X$: scan from bucket $X \mod 10$ until empty bucket.
Collision Handling 1

insert $X$ $O(1)$ average
remove $X$ $O(1)$ average
exists $X$ $O(1)$ average

This kind of collision handling is called *Open Addressing*

Variants

- Linear probing
- Quadratic probing
Collision Handling 2

Store all collisions in linked lists.

insert 12

insert 34

insert 22
Collision Handling 2

**insert** $X$  
add item to the linked list at bucket $X \mod 10$  
$O(1)$ average

**remove** $X$  
remove item in the linked list at bucket $X \mod 10$  
$O(1)$ average

**exists** $X$  
scan the linked list at bucket $X \mod 10$  
$O(1)$ average

This kind of collision handling is called *Separate Chaining*
Hash Table

The data structure that we built is called *Hash Table*.

The core function that we used

\[ h(X) = X \mod P \]

is called the *hash function*, which maps *something* into a range of integers that we can handle.
The data structure that we built is called *Hash Table*.

The core function that we used

\[ h(X) = X \mod P \]

is called the *hash function*, which maps *something* into a range of integers that we can handle.

Is it better to use a prime number for \( P \)?
The performance of a hash table is determined by the load factor $N/K$, where $N$ is the number of items and $K$ is the number of buckets.

For *Separate Chaining*, length of linked lists is proportional to load factor

<table>
<thead>
<tr>
<th>Load factor</th>
<th>Wasted space</th>
<th>Performance</th>
</tr>
</thead>
<tbody>
<tr>
<td>smaller</td>
<td>smaller or larger?</td>
<td>faster or slower?</td>
</tr>
<tr>
<td>larger</td>
<td>smaller or larger?</td>
<td>faster or slower?</td>
</tr>
</tbody>
</table>
Hash Table

The performance of a hash table is determined by the load factor $\frac{N}{K}$, where $N$ is the number of items and $K$ is the number of buckets.

For *Separate Chaining*, length of linked lists is proportional to load factor.

<table>
<thead>
<tr>
<th>Load factor</th>
<th>Wasted space</th>
<th>Performance</th>
</tr>
</thead>
<tbody>
<tr>
<td>smaller</td>
<td>smaller or larger?</td>
<td>faster or slower?</td>
</tr>
<tr>
<td>larger</td>
<td>smaller or larger?</td>
<td>faster or slower?</td>
</tr>
</tbody>
</table>
Hash Table

The performance of a hash table is determined by the load factor $\frac{N}{K}$, where $N$ is the number of items and $K$ is the number of buckets.

For *Separate Chaining*, length of linked lists is proportional to load factor.

<table>
<thead>
<tr>
<th>Load factor</th>
<th>Wasted space</th>
<th>Performance</th>
</tr>
</thead>
<tbody>
<tr>
<td>smaller</td>
<td>smaller or larger?</td>
<td>faster or slower?</td>
</tr>
<tr>
<td>larger</td>
<td>smaller or larger?</td>
<td>faster or slower?</td>
</tr>
</tbody>
</table>
Hash Table

The performance of a hash table is determined by the load factor \( \frac{N}{K} \), where \( N \) is the number of items and \( K \) is the number of buckets.

For *Separate Chaining*, length of linked lists is proportional to load factor

<table>
<thead>
<tr>
<th>Load factor</th>
<th>Wasted space</th>
<th>Performance</th>
</tr>
</thead>
<tbody>
<tr>
<td>smaller</td>
<td>smaller or larger?</td>
<td>faster or slower?</td>
</tr>
<tr>
<td>larger</td>
<td>smaller or larger?</td>
<td>faster or slower?</td>
</tr>
</tbody>
</table>
C++ Standard Template Library

- C++11 only

```c++
#include <unordered_set>
using namespace std;
...
unordered_set<int> s;
s.insert(123);
int x = s.count(123); //1
s.erase(123);
```

```c++
#include <unordered_map>
using namespace std;
...
unordered_map<int, int> m;
m[123] = 456;
int x = m[123]; //456
m.erase(123);
```
Problem Description

\(N\) operations:
- insert \(X\)
- get min
  (print the minimum item)
- remove min
  (remove the minimum item)

Constraints:
- \(1 \leq N \leq 10000\)
- \(1 \leq X \leq 10^{18}\)

Input

6
insert 12
insert 74
get min
insert 39
remove min
get min

Output

12
39
Binary Heap

*Shape property*: complete binary tree

- The bottom level is filled from left to right
- All above levels are fully filled
Binary Heap

*Shape property:* complete binary tree
  - The bottom level is filled from left to right
  - All above levels are fully filled

*Heap property:* every node $\geq$ its parent
Binary Heap

*Shape property*: complete binary tree

- The bottom level is filled from left to right
- All above levels are fully filled

*Heap property*: every node $\geq$ its parent

Are all root-to-leaf paths sorted?
Binary Heap

- Number of nodes \( N \)
- Tree height \( h \) = maximum number of edges from root to leaf
- \( 2^h \leq N \leq 2^{h+1} - 1 \)
- \( h \leq \log_2 N = O(\log N) \)
- What are the possible lengths of all root-to-leaf paths?
Binary Heap

- Number of nodes = \( N \)
- Tree height \( h = \) maximum number of edges from root to leaf
- \( 2^h \leq N \leq 2^{h+1} - 1 \)
- \( h \leq \log_2 N = O(\log N) \)
- What are the possible lengths of all root-to-leaf paths?
- either \( h \) or \( h - 1 \)
Binary Heap

insert 2

- Add new node at the bottom
Binary Heap

insert 2

- Add new node at the bottom
- Heap property violated
- How many root-to-leaf paths are not sorted?
Binary Heap

**insert 2**

- Add new node at the bottom
- Heap property violated
- How many root-to-leaf paths are not sorted?
- Only one, let’s (merge/quick/bubble/insertion/selection) sort that path
Binary Heap

insert 2

- Add new node at the bottom
- Heap property violated
- How many root-to-leaf paths are not sorted?
- Only one, let’s insertion sort that path
Binary Heap

insert 2

- Add new node at the bottom
- Heap property violated
- How many root-to-leaf paths are not sorted?
- We call it *sift-up*
Binary Heap

insert 2

- Add new node at the bottom
- Heap property violated
- How many root-to-leaf paths are not sorted?
- We call it *sift-up*

![Binary Heap Diagram]

smaller than parent, swap
Binary Heap

insert 2

- Add new node at the bottom
- Heap property violated
- How many root-to-leaf paths are not sorted?
- We call it *sift-up*
Binary Heap

insert 2

► Add new node at the bottom
► Heap property violated
► How many root-to-leaf paths are not sorted?
► We call it *sift-up*
Binary Heap

insert 2

- Add new node at the bottom
- Heap property violated
- How many root-to-leaf paths are not sorted?
- We call it *sift-up*
Binary Heap

insert 2

- Add new node at the bottom
- Heap property violated
- How many root-to-leaf paths are not sorted?
- We call it *sift-up*

larger than parent, stop
Binary Heap

insert 2

- Add new node at the bottom
- Heap property violated
- How many root-to-leaf paths are not sorted?
- We call it *sift-up*
- Done?
Binary Heap

Proof

- The path on which we performed sift-up must be sorted
- Sift-up modified the prefixes of some other root-to-leaf paths
- Those prefixes must be sorted
- Those prefixes could only become smaller after sift-up
- So, all root-to-leaf paths are sorted
Binary Heap

Time complexity

- Maximum number of swaps = $h = O(\log N)$
Binary Heap

get min
- Simply return root
- $O(1)$
Binary Heap

remove min

- Just remove it?

```
1
  2
  8
    8
    9
  5
    7
    6
  3
    5
    4
  7
```
Binary Heap

remove min

- Just remove it?
- Shape property violated
- Let’s replace it with the bottom node
Binary Heap

remove min

- Heap property violated
- How many root-to-leaf paths are not sorted?
Binary Heap

remove min

- Heap property violated
- How many root-to-leaf paths are not sorted?
- Perform *sift-down*
Binary Heap

remove min

- Heap property violated
- How many root-to-leaf paths are not sorted?
- Perform *sift-down*

greater than both children; swap with the smaller
Binary Heap

- remove min
  - Heap property violated
  - How many root-to-leaf paths are not sorted?
  - Perform *sift-down*
Binary Heap

remove min

- Heap property violated
- How many root-to-leaf paths are not sorted?
- Perform *sift-down*

greater than left child only; swap
Binary Heap

remove min

- Heap property violated
- How many root-to-leaf paths are not sorted?
- Perform sift-down
Binary Heap

remove min

- Heap property violated
- How many root-to-leaf paths are not sorted?
- Perform *sift-down*

![Binary Heap Diagram]

no smaller children; stop
Binary Heap

remove min
- Heap property violated
- How many root-to-leaf paths are not sorted?
- Perform *sift-down*
- Done?
Binary Heap

Proof

- The path on which we performed sift-down must be sorted
- Sift-down modified the *prefixes* of some root-to-leaf paths
- Those *prefixes* must be sorted
- Those *prefixes* could not become larger than the postfixes
- So, all root-to-leaf paths are sorted
Binary Heap

Time complexity

- Maximum number of swaps = \( h = O(\log N) \)
Binary Heap

Implementation

- plain array
Binary Heap

Implementation

- plain array
Binary Heap

Implementation
- plain array

- parent $i$
- left child $2i$
- right child $2i + 1$
Heapsort

1. Build heap naively: \( O(N \log N) \)
2. Repeatedly remove min: \( O(N \log N) \)

Any similarity between heapsort and selection sort?
Heapsort

1. Build heap naively: $O(N \log N)$
2. Repeatedly remove min: $O(N \log N)$

Any similarity between heapsort and selection sort?

Heapsort vs Quicksort

- In reality, the computer reads not only one memory slot at a time, but a chunk of contiguous memory into cache
- Heapsort does not operate on contiguous memory, introducing cache miss
- Quicksort operate on contiguous memory, benefits from cache
Build binary heap in linear time

- Create the shape first
- Add elements in the lowest levels first
- Perform sift-downs

3  2  1  4  6  7
Build binary heap in linear time

- Create the shape first
- Add elements in the lowest levels first
- Perform sift-downs

| 2 | 1 | 4 | 6 | 7 |

```
  3
```

```
    3
   /   \
 1     4
 |
2
```
Build binary heap in linear time

- Create the shape first
- Add elements in the lowest levels first
- Perform sift-downs

```
1  4  6  7
```

![Binary heap diagram with elements 1, 4, 6, 7 and nodes 2 and 3]
Build binary heap in linear time

- Create the shape first
- Add elements in the lowest levels first
- Perform sift-downs

```
4 6 7
```

```
1
  2
  3
```
Build binary heap in linear time

- Create the shape first
- Add elements in the lowest levels first
- Perform sift-downs

6 7

greater than left child; swap
Build binary heap in linear time

- Create the shape first
- Add elements in the lowest levels first
- Perform sift-downs

```
6 7
```

```
1 2
```

```
4
```

```
3
```
Build binary heap in linear time

- Create the shape first
- Add elements in the lowest levels first
- Perform sift-downs

```
7
```

greater than left child; swap
Build binary heap in linear time

- Create the shape first
- Add elements in the lowest levels first
- Perform sift-downs
Build binary heap in linear time

- Create the shape first
- Add elements in the lowest levels first
- Perform sift-downs

```
greater than left child; swap
```
Build binary heap in linear time

- Create the shape first
- Add elements in the lowest levels first
- Perform sift-downs
Build binary heap in linear time

- Create the shape first
- Add elements in the lowest levels first
- Perform sift-downs

![Binary heap diagram]

1. greater than right child; swap
Build binary heap in linear time

- Create the shape first
- Add elements in the lowest levels first
- Perform sift-downs
Build binary heap in linear time

- Create the shape first
- Add elements in the lowest levels first
- Perform sift-downs
Build binary heap in linear time

<table>
<thead>
<tr>
<th>Layer</th>
<th>#swaps</th>
<th>#nodes</th>
</tr>
</thead>
<tbody>
<tr>
<td>1st</td>
<td>$h$</td>
<td>1</td>
</tr>
<tr>
<td>2nd</td>
<td>$h - 1$</td>
<td>2</td>
</tr>
<tr>
<td>3rd</td>
<td>$h - 2$</td>
<td>4</td>
</tr>
<tr>
<td>...</td>
<td></td>
<td></td>
</tr>
<tr>
<td>$h^{th}$</td>
<td>1</td>
<td>$2^{h-1}$</td>
</tr>
<tr>
<td>$(h + 1)^{th}$</td>
<td>0</td>
<td>$2^h$</td>
</tr>
</tbody>
</table>

(top) (bottom)
Build binary heap in linear time

<table>
<thead>
<tr>
<th>layer</th>
<th>#swaps</th>
<th>#nodes</th>
</tr>
</thead>
<tbody>
<tr>
<td>1ˢᵗ</td>
<td>$h$</td>
<td>1</td>
</tr>
<tr>
<td>2ⁿᵈ</td>
<td>$h - 1$</td>
<td>2</td>
</tr>
<tr>
<td>3ʳᵈ</td>
<td>$h - 2$</td>
<td>4</td>
</tr>
<tr>
<td>...</td>
<td></td>
<td></td>
</tr>
<tr>
<td>$h^{th}$</td>
<td>1</td>
<td>$2^{h-1}$</td>
</tr>
<tr>
<td>(bottom)</td>
<td>$(h + 1)^{th}$</td>
<td>0</td>
</tr>
</tbody>
</table>

Total #swaps = $\sum_{i=0}^{h} 2^{h-i} i = 2^h \sum_{i=0}^{h} \frac{i}{2^i} \leq 2^h \sum_{i=0}^{\infty} \frac{i}{2^i} = 2^h (2) = O(N)$

because

$$\sum_{i=0}^{\infty} \frac{i}{2^i} = \sum_{i=0}^{\infty} \frac{i + 1}{2^{i+1}} = \sum_{i=0}^{\infty} \frac{i}{2^{i+1}} + \sum_{i=0}^{\infty} \frac{1}{2^{i+1}} = \frac{1}{2} \sum_{i=0}^{\infty} \frac{i}{2^i} + 1$$

$\Rightarrow \sum_{i=0}^{\infty} \frac{i}{2^i} = 2$
Build binary heap in linear time

- Time complexity: $O(N)$
- Does that mean we can sort an array in $O(N)$ time?
Build binary heap in linear time

- Time complexity: $O(N)$
- Does that mean we can sort an array in $O(N)$ time?
- No, removing all minimum still need $O(N \log N)$
Merging binary heaps

- Can we merge two binary heaps in $O(N \log N)$ time?
Merging binary heaps

- Can we merge two binary heaps in $O(N \log N)$ time? insert all elements from one to another
- Can we merge two binary heaps in $O(N)$ time?
Merging binary heaps

- Can we merge two binary heaps in $O(N \log N)$ time? Insert all elements from one to another.
- Can we merge two binary heaps in $O(N)$ time? Concatenate two arrays and build heap.
- Can we merge faster?
- Yes, but very complicated.
- Better use alternatives such as Leftist heap or Binomial heap.
The afore introduced binary heap was min-heap

`priority_queue` is max-heap

```cpp
#include <queue>
...
std::priority_queue<int> q;
q.push(4); q.push(7); q.push(3);
int x = q.top(); // x = 7
q.pop(); x = q.top(); // x = 4
```
Problem Description

\( N \) operations:
- insert \( X \)
- exists \( X \)
- remove \( X \)
- get min

Constraints:
- \( 1 \leq N \leq 10000 \)
- \( 1 \leq X \leq 10^{18} \)

Input

6
insert 12
insert 39
insert 74
exists 74
remove min
get min

Output

yes
39
Binary Search Tree

Binary tree
Every node > its left descendants
Every node < its right descendants

Do we get a sorted array in
- pre-order traversal?
- in-order traversal?
- post-order traversal?
Binary Search Tree

exists 6

▶ Binary search from root
Binary Search Tree

exists 6

- Binary search from root

6 is not here; go right
Binary Search Tree

exists 6
  ▶ Binary search from root

6 is not here; go left
Binary Search Tree

exists 6

- Binary search from root

6 is here; done
Binary Search Tree

exists 6

- Binary search from root
- Each iteration we discard half of the subtree
- Time complexity $= O(\log N)$ in average

```
  5
 / \   \\
1   8
 /   /   \\
3   6   9
 /     /     \\
4     7     
```
Binary Search Tree

- Binary search from root
- Each iteration we discard half of the subtree
- Time complexity = $O(\log N)$ in average
- Time complexity = $O(N)$ in worst case
Binary Search Tree

insert 2
  - Binary search a place to insert

```
  5
 /   \
1     8
|     |
3     6
|     |
4     7
|     |
```
Binary Search Tree

insert 2

- Binary search a place to insert

```
      5
     / \
   1    8
  /     / \
 3     6    9
 /     /   /
4     7    
```

go left
Binary Search Tree

insert 2

- Binary search a place to insert
Binary Search Tree

insert 2

- Binary search a place to insert

diagram:

- Root: 5
  - Left child: 3
    - Left child: 4
    - Right child: 6
      - Right child: 7
  - Right child: 8
    - Right child: 9
Binary Search Tree

insert 2

- Binary search a place to insert
Binary Search Tree

insert 2

- Binary search a place to insert
- Each iteration we discard half of the subtree
- Time complexity $= O(\log N)$ in average
Binary Search Tree

insert 2

- Binary search a place to insert
- Each iteration we discard half of the subtree
- Time complexity = $O(\log N)$ in average
- Time complexity = $O(N)$ in worst case
Binary Search Tree

remove 8
remove 5

- Binary search the node
- Replace it with the greatest node less than it

Time complexity = $O(\log N)$ in average

Time complexity = $O(N)$ in worst case
Binary Search Tree

- remove 8
- remove 5
  - Binary search the node
  - Replace it with the greatest node less than it

```
[9x251]Binary Search Tree
[28x236]remove 8
[28x222]remove 5
  ▶ Binary search the node
  ▶ Replace it with the greatest node less than it
```
Binary Search Tree

remove 8
remove 5

- Binary search the node
- Replace it with the greatest node less than it
Binary Search Tree

- remove 8
- remove 5
  - Binary search the node
  - Replace it with the greatest node less than it

Time complexity = $O(\log N)$ in average

Time complexity = $O(N)$ in worst case

replace 8 with 7
Binary Search Tree

remove 8
remove 5
  ▶ Binary search the node
  ▶ Replace it with the greatest node less than it
Binary Search Tree

remove 8
remove 5
  - Binary search the node
  - Replace it with the greatest node less than it
Binary Search Tree

- remove 8
- remove 5
  - Binary search the node
  - Replace it with the greatest node less than it
Binary Search Tree

- remove 8
- remove 5
  - Binary search the node
  - Replace it with the greatest node less than it
Binary Search Tree

- remove 8
- remove 5
  - Binary search the node
  - Replace it with the greatest node less than it

Time complexity = $O(\log N)$ in average

Time complexity = $O(N)$ in worst case

replace 5 with 4
Binary Search Tree

- remove 8
- remove 5
  - Binary search the node
  - Replace it with the greatest node less than it
  - Locate the node = $O(\log N)$ in average
  - Locate the smaller node = $O(\log N)$ in average
  - Time complexity = $O(\log N)$ in average
  - Time complexity = $O(N)$ in worst case
Augmented Binary Search Tree

- We can attach values to nodes, e.g.
  - existence

- Nodes are sorted by their keys
- How does it affect insert and remove?
- Do we still need remove?
Augmented Binary Search Tree

- We can attach values to nodes, e.g.
  - existence
  - subtree size
- Nodes are sorted by their keys
- How does it affect insert and remove?
- Can we find the $k^{th}$ largest element?
Binary Search Tree

- It is easy to generate a worst case testcase
- How to get rid of the $O(N)$ worst case?
- Use variants of BST
  - AVL tree
  - Red-black tree
  - AA tree
  - Treap
  - Size-balanced tree
  - Splay tree
Implementation

```c
int root, value[10000], left[10000], right[10000];
```
or, if you know C++ and pointer very well,

```cpp
struct BST {
    int value;
    BST *left, *right;
} *root, memory[10000], memory_i;
```

Avoid `malloc()` because it is slow
C++ Standard Template Library

- set, map, multiset, multimap
- Red-black tree internally
- worst case $O(\log N)$

```cpp
#include <set>
...
std::set<int> s;
s.insert(456);
int x = s.count(456); //1
x.erase(456);

#include <map>
...
std::map<int, int> m;
m[456] = 123;
int x = m[456]; //123
m.erase(456);
```
Practice problems

- 01090 Diligent
- 01019 Addition II
- 30107 What is the Median?
- M0811 Alice’s Bookshelf
- M0913 ls
- AP121 Dispatching
Two literal meanings:
- Insightful merging
- Open-up merging (not this one)

Similar to *Union by Rank* in disjoint-set union (Data Structures (III))

Seems there is not a commonly used English term

Problem: need to merge $N$ items from small binary heaps into a large binary heap

Idea: when merging two heaps, insert the smaller one into the larger one

Time complexity: $O(N \log^2 N)$
Consider an item $X$

Initially $X$ belongs to a heap of size 1

Whenever the heap containing $X$ is merged into a larger heap, the size of the heap is at least doubled

So $X$ can be inserted into another heap for at most $O(\log N)$ times

Each of the $N$ items can be inserted into another heap for at most $O(\log N)$ times

Total number of heap insertions = $O(N \log N)$

Time complexity = $O(N \log^2 N)$