Outline

1. Asymptotic Analysis
2. Sorting
3. Searching
Notations

- $\Theta$, $O$, $o$, $\Omega$, and $\omega$.

- Similar to $=$, $\leq$, $<$, $\geq$, and $>$. 
- \( \Theta, O, o, \Omega, \) and \( \omega \).

- Similar to \( =, \leq, <, \geq, \) and \( > \).
  - But not exactly the same, e.g., \( 2n^2 > n^2 \) but \( 2n^2 = \Theta(n^2) \).
How to compare two growth rates?

Idea 1: using definition, e.g., find the two coefficients $c_1$, $c_2$, and the proper $n_0$ for proving $\Theta$.

Idea 2: limit test, may need L'Hospital rule in some case.
How to compare two growth rates?

- Idea 1: using definition, e.g., find the two coefficients $c_1$, $c_2$, and the proper $n_0$ for proving $\Theta$. 
How to compare two growth rates?

- Idea 1: using definition, e.g., find the two coefficients $c_1$, $c_2$, and the proper $n_0$ for proving $\Theta$.

- Idea 2: limit test, may need L’Hospital rule in some case.
Outline

1. Asymptotic Analysis
2. Sorting
3. Searching
Basic idea: scan the array $A$ from left to right, and each $i$-th step ($2 \leq i \leq n$):

- Find the proper position for the $i$-th number in $A_{[1, \ldots, i]}$, and then insert it.
- You can use either linear scanning or binary search on the first $i-1$ numbers.

Time complexity is $\Theta(n^2)$ in the worst case, and (extra) space complexity is $\Theta(1)$. 
Basic idea: scan the array $A$ from left to right, and each $i$-th step ($2 \leq i \leq n$):

- find the proper position for the $i$-th number in $A[1, \cdots, i]$, and then insert it.
**Insertion sort**

- **Basic idea:** scan the array $A$ from left to right, and each $i$-th step ($2 \leq i \leq n$):
  - find the proper position for the $i$-th number in $A[1, \cdots, i]$, and then insert it.
  - You can use either linear scanning or binary search on the first $i - 1$ numbers.

Time complexity is $\Theta(n^2)$ in the worst case, and (extra) space complexity is $\Theta(1)$. 

© Hu Ding (Michigan State University)
Insertion sort

- Basic idea: scan the array $A$ from left to right, and each $i$-th step $(2 \leq i \leq n)$:
  - find the proper position for the $i$-th number in $A[1, \cdots, i]$, and then insert it.
  - You can use either linear scanning or binary search on the first $i - 1$ numbers.

- Time complexity is $\Theta(n^2)$ in the worst case, and (extra) space complexity is $\Theta(1)$. 
Merge sort

- Basic idea: divide-and-conquer
  - Divide an $n$-number array into two equal sub-arrays, and recursively sort both of them; finally merge them to form the final solution.
  - $\Theta(n \log n)$ time complexity and $\Theta(n)$ extra space complexity. Easy to be parallel in practice.
Merge sort

- Basic idea: divide-and-conquer
  - Divide an $n$-number array into two equal sub-arrays, and recursively sort both of them; finally merge them to form the final solution.
  - $\Theta(n \log n)$ time complexity and $\Theta(n)$ extra space complexity. Easy to be parallel in practice.

- Master theorem for calculating the running time for general divide-and-conquer algorithms.

### Master Theorem ($T(n) = aT(n/b) + f(n)$)

1. If $f(n) = O(n^{\log_b a - \epsilon})$ for some constant $\epsilon > 0$, then $T(n) = \Theta(n^{\log_b a})$.
2. If $f(n) = \Theta(n^{\log_b a})$, then $T(n) = \Theta(n^{\log_b a} \log n)$.
3. If $f(n) = \Omega(n^{\log_b a + \epsilon})$ for some constant $\epsilon > 0$, and $af(n/b) \leq cf(n)$ for some $c < 1$ for sufficiently large $n$, then $T(n) = \Theta(f(n))$. 
Heap sort

- How to build a heap with linear time and constant extra space complexity.

Use heap to sort:
- Each time output the max/min number from the root;
- Replace the root by the final number of the current heap, and do max/min heapify.
Heap sort

- How to build a heap with linear time and constant extra space complexity.
- Use heap to sort:
Heap sort

- How to build a heap with linear time and constant extra space complexity.

- Use heap to sort:
  - each time output the max/min number from the root;
Heap sort

- How to build a heap with linear time and constant extra space complexity.

- Use heap to sort:
  - each time output the max/min number from the root;
  - replace the root by the final number of the current heap, and do max/min heapify.
Outline

1. Asymptotic Analysis
2. Sorting
3. Searching
Simple BST: no bound for the height.
Binary search tree

- Simple BST: no bound for the height.
- AVL-Tree: balanced BST, and the height is $\Theta(\log n)$:
Binary search tree

- Simple BST: no bound for the height.
- AVL-Tree: balanced BST, and the height is $\Theta(\log n)$:
  - Four basic operations: LR, RR, LR+RR, RR+LR.
About the implementation for the basic operations, take RR as an example:

1. \( y \rightarrow \text{parent} = x \rightarrow \text{parent}; \)
2. \( x \rightarrow \text{lchild} = y \rightarrow \text{rchild}; \)
3. \( y \rightarrow \text{rchild} \rightarrow \text{parent} = x; \)
4. \( y \rightarrow \text{rchild} = x; \)
5. \( x \rightarrow \text{parent} = y; \)
Hashing

- Two types: chaining and open addressing.
Hashing

- Two types: chaining and open addressing.
- Search time: expected $\Theta(1)$, but could be as bad as $\Theta(n)$ in the worst case.

Dynamically updating: Double the hash table and re-hash all the elements when $n$ is bigger than the size $m$. Suppose the initial size of the hash table is $m_0$, update $t$ times; then the size becomes $m_0 \rightarrow 2m_0 \rightarrow 2^2m_0 \rightarrow \cdots \rightarrow 2^t m_0$; the current number of elements $n = 2^t m_0$.

So the total re-hash time is $T = \sum_{j=0}^{t} 2^j m_0 = \Theta(n)$, i.e., $T/n = \Theta(1)$.

When $n$ is too small, we need to shrink the hash table. If we reduce $m$ to be $m/2$ when $n = m/2$, imagine we have a sequence of operations like insertion, deletion, insertion, deletion, insertion, ......, we will alternatively double and shrink the hash table for $n = m/2$ and $n = m/2 + 1$, which is terrible.

So we reduce $m$ to be $m/2$ when $n = m/\alpha$, where $\alpha > 2$. 

© Hu Ding (Michigan State University)
Hashing

- Two types: chaining and open addressing.
- Search time: expected $\Theta(1)$, but could be as bad as $\Theta(n)$ in the worst case.
- Dynamically updating:
Hashing

- Two types: chaining and open addressing.
- Search time: expected $\Theta(1)$, but could be as bad as $\Theta(n)$ in the worst case.
- Dynamically updating:
  - Double the hash table and re-hash all the elements when $n$ is bigger than the size $m$. 

$\text{Suppose the initial size of the hash table is } m_0, \text{ update } t \text{ times; then the size becomes } m_0 \rightarrow 2m_0 \rightarrow 2^2m_0 \rightarrow \cdots \rightarrow 2^t m_0; \text{ the current number of elements } n = 2^t m_0.$

$\text{So the total re-hash time is } T = \sum_{j=0}^{t} 2^j m_0 = \Theta(n), \text{ i.e., } T/n = \Theta(1).$

When $n$ is too small, we need to shrink the hash table.
If we reduce $m$ to be $m/2$ when $n = m/2$, imagine we have a sequence of operations like insertion, deletion, insertion, deletion, insertion, ......, we will alternatively double and shrink the hash table for $n = m/2$ and $n = m/2 + 1$, which is terrible.
So we reduce $m$ to be $m/2$ when $n = m/\alpha$, where $\alpha > 2$. 

© Hu Ding (Michigan State University)  CSE 331 Algorithm and Data Structures  11 / 11
Hashing

- Two types: chaining and open addressing.
- Search time: expected $\Theta(1)$, but could be as bad as $\Theta(n)$ in the worst case.
- Dynamically updating:
  - Double the hash table and re-hash all the elements when $n$ is bigger than the size $m$.
  - Suppose the initial size of the hash table is $m_0$, update $t$ times; then the size becomes $m_0 \rightarrow 2m_0 \rightarrow 2^2m_0 \rightarrow \cdots \rightarrow 2^t m_0$; the current number of elements $n = 2^t m_0$. 

So the total re-hash time is $T = \sum_{j=0}^{t} 2^j m_0 = \Theta(n)$, i.e., $T/n = \Theta(1)$.

When $n$ is too small, we need to shrink the hash table. If we reduce $m$ to be $m/2$ when $n = m/2$, imagine we have a sequence of operations like insertion, deletion, insertion, deletion, insertion, ......, we will alternatively double and shrink the hash table for $n = m/2$ and $n = m/2 + 1$, which is terrible.

So we reduce $m$ to be $m/2$ when $n = m/\alpha$, where $\alpha > 2$. 
Hashing

- Two types: chaining and open addressing.
- Search time: expected $\Theta(1)$, but could be as bad as $\Theta(n)$ in the worst case.
- Dynamically updating:
  - Double the hash table and re-hash all the elements when $n$ is bigger than the size $m$.
  - Suppose the initial size of the hash table is $m_0$, update $t$ times; then the size becomes $m_0 \rightarrow 2m_0 \rightarrow 2^2m_0 \rightarrow \cdots \rightarrow 2^t m_0$; the current number of elements $n = 2^t m_0$.
  - So the total re-hash time is $T = \sum_{j=0}^{t} 2^j m_0 = \Theta(n)$, i.e., $T/n = \Theta(1)$. 
Hashing

- Two types: chaining and open addressing.
- Search time: expected $\Theta(1)$, but could be as bad as $\Theta(n)$ in the worst case.
- Dynamically updating:
  - Double the hash table and re-hash all the elements when $n$ is bigger than the size $m$.
  - Suppose the initial size of the hash table is $m_0$, update $t$ times; then the size becomes $m_0 \rightarrow 2m_0 \rightarrow 2^2m_0 \rightarrow \cdots \rightarrow 2^tm_0$; the current number of elements $n = 2^tm_0$.
  - So the total re-hash time is $T = \sum_{j=0}^{t} 2^jm_0 = \Theta(n)$, i.e., $T/n = \Theta(1)$.
  - When $n$ is too small, we need to shrink the hash table.
Hashing

- Two types: chaining and open addressing.
- Search time: expected $\Theta(1)$, but could be as bad as $\Theta(n)$ in the worst case.
- Dynamically updating:
  - Double the hash table and re-hash all the elements when $n$ is bigger than the size $m$.
  - Suppose the initial size of the hash table is $m_0$, update $t$ times; then the size becomes $m_0 \rightarrow 2m_0 \rightarrow 2^2m_0 \rightarrow \cdots \rightarrow 2^t m_0$; the current number of elements $n = 2^t m_0$.
  - So the total re-hash time is $T = \sum_{j=0}^{t} 2^j m_0 = \Theta(n)$, i.e., $T/n = \Theta(1)$.
  - When $n$ is too small, we need to shrink the hash table.
  - If we reduce $m$ to be $m/2$ when $n = m/2$, imagine we have a sequence of operations like insertion, deletion, insertion, deletion, insertion, ...., we will alternatively double and shrink the hash table for $n = m/2$ and $n = m/2 + 1$, which is terrible.
Hashing

- Two types: chaining and open addressing.
- Search time: expected $\Theta(1)$, but could be as bad as $\Theta(n)$ in the worst case.
- Dynamically updating:
  - Double the hash table and re-hash all the elements when $n$ is bigger than the size $m$.
  - Suppose the initial size of the hash table is $m_0$, update $t$ times; then the size becomes $m_0 \rightarrow 2m_0 \rightarrow 2^2m_0 \rightarrow \cdots \rightarrow 2^tm_0$; the current number of elements $n = 2^tm_0$.
  - So the total re-hash time is $T = \sum_{j=0}^{t} 2^jm_0 = \Theta(n)$, i.e., $T/n = \Theta(1)$.
  - When $n$ is too small, we need to shrink the hash table.
  - If we reduce $m$ to be $m/2$ when $n = m/2$, imagine we have a sequence of operations like insertion, deletion, insertion, deletion, insertion, ...., we will alternatively double and shrink the hash table for $n = m/2$ and $n = m/2 + 1$, which is terrible.
  - So we reduce $m$ to be $m/2$ when $n = m/\alpha$, where $\alpha > 2$. 

©Hu Ding (Michigan State University)