Introduction

Asymptotic Algorithm Analysis

Running time is expressed as $T(n)$ for some function T on input size n.

Best, Worst, Average Cases

- Best case: least number of steps required, corresponding to the **ideal input**
- Worst case: most number of steps required, corresponding to the **most difficult input**.
- Average case: average number of steps required, over **purely random inputs**.

Big-Oh

Definition: A non-negatively valued function, $T(n)$, is in the set $O(f(n))$ if there exist two positive constants c and n_0 such that
$T(n) \leq cf(n)$ for all $n > n_0$.

upper bound

- Rule 1: If$f(n) = O(g(n))$, then $cf(n) = O(g(n))$.
- Rule 2: If $f_1(n) = O\left(g_1(n)\right)$ and $f_2(n) = O\left(g_2(n)\right)$, then $f_1(n) + f_2(n) = O\left(\max\left\{g_1(n), g_2(n)\right\}\right)$.
- Rule 3: If $f_1(n) = O\left(g_1(n)\right)$ and $f_2(n) = O\left(g_2(n)\right)$, then $f_1(n) \cdot f_2(n) = O\left(g_1(n) \cdot g_2(n)\right)$.
- Rule 4: If $f(n) = O(g(n))$ and $g(n) = O(h(n))$, then $f(n) = O(h(n))$.

Big-Omega

Definition: For $T(n)$ a non-negatively valued function, $T(n)$ is in the set $\Omega(g(n))$ if there exist two positive constants C and n_0 such that $T(n) \geq cg(n)$ for all $n > n_0$.

lower bound

Theta Notation

When big-oh and big-omega coincide, we indicate this by using big-theta (Θ) notation.

Comparison Sort

General concept

	Worst Case Time	Average Case Time	In Place	Stable
Insertion	$O(N^2)$	$O(N^2)$	Yes	Yes
Selection	$O(N^2)$	$O(N^2)$	Yes	No
Bubble	$O(N^2)$	$O(N^2)$	Yes	Yes
Merge Sort	$O(N\log N)$	$O(N\log N)$	No	Yes
Quick Sort	$O(N^2)$	$O(N\log N)$	Weakly	No

Insertion Sort

```
template <typename T, typename Compare = std::less<T>>
void insertion_sort(std::vector<T> &vector, Compare comp = Compare()) {
    size_t size = vector.size();
    // Insert A[i] to the correct location in the sorted part
    for (size_t i = 1; i < size; i++) {
        size_t j = 0;
        while (j < i && !comp(vector[i], vector[j])) {
            // Find the location to insert the value
            // Use >= to ensure stablity (since we traverse from left to right)
            j++;
        }
        const T tmp = vector[i]; // Store the value we need to insert
        vector.erase(vector.begin() + i);
        vector.insert(vector.begin() + j, tmp);
    }
}
```

The **best case** time complexity is $O(N)$. It happens when the **array is already sorted**.

Features:

1. Unsorted part remains original order
2. Sorted part is not the final position

Selection Sort

```
template <typename T, typename Compare = std::less<T>>
void selection_sort(std::vector<T> &vector, Compare comp = Compare()) {
    // base case
    size_t size = vector.size();
    if (size <= 1) {
        return;
    }
    // when size > 1
    for (size_t i = 0; i < size - 1; i++) {
        // find the smallest element in the unsorted part
        size_t smallest = i;
        for (size_t j = i + 1; j < size; j++) {
            if (comp(vector[j], vector[smallest])) {
                smallest = j;
```

```
            }
        }
        // swap it with the current element
        swap_val(vector, i, smallest);
    }
}
```

Features:

1. Sorted part is in final position

Bubble Sort

```
template <typename T, typename Compare = std::less<T>>
void bubble_sort(std::vector<T> &vector, Compare comp = Compare()) {
    // base case
    size_t size = vector.size();
    if (size <= 1) {
        return;
    } else if (size == 2) {
        if (comp(vector[0], vector[1])) {
            swap_val(vector, 0, 1);
        }
        return;
    }
    // when size > 2
    for (size_t i = size - 2; i > 0; i--) {
        // for each element, compares two adjacent items and swap them to keep
them in ascending order, until the current element
        for (size_t j = 0; j <= i; j++) {
            if (comp(vector[j + 1], vector[j])) {
                swap_val(vector, j, j + 1);
            }
        }
    }
}
```

Features:

1. The biggest element keeps popping to the right, so the sorted part is in the final position

Merge Sort

```
template <typename T, typename Compare = std::less<T>>
void merge(std::vector<T> &left, std::vector<T> &right, std::vector<T> &vector,
Compare comp = Compare()) {
    // merge two parts and ensure correct order
    size_t i = 0, j = 0, k = 0;
    while (i != left.size() && j != right.size()) {
        if (!comp(right.at(j), left.at(i))) {
            vector[k] = left.at(i);
            i++;
        } else {
            vector[k] = right.at(j);
            j++;
        }
```

```
        k++;
    }
    if ((left.begin() + i) == left.end()) {
        for (j = j; j < right.size(); j++) {
            vector[k] = right.at(j);
            k++;
        }
    } else {
        for (i = i; i < left.size(); i++) {
            vector[k] = left.at(i);
            k++;
        }
    }
}

template <typename T, typename Compare = std::less<T>>
void merge_sort(std::vector<T> &vector, Compare comp = Compare())v{
    // base case
    if (!comp(vector.back(), vector.front()) && vector.size() <= 2) {
        return;
    }
    // divide
    size_t mid = vector.size() / 2;
    std::vector<T> left(vector.begin(), vector.begin() + mid);
    std::vector<T> right(vector.begin() + mid, vector.end());
    // sort the two parts
    merge_sort(left, comp);
    merge_sort(right, comp);
    // merge
    merge(left, right, vector, comp);
}
```

Features:

1. First left sub parts, then right sub parts
2. Sub parts are sorted

Quick Sort

```
template <typename T, typename Compare = std::less<T>>
void find_ij(std::vector<T> &vector, size_t begin, size_t end, const T p_val,
size_t &i, size_t &j, Compare comp = Compare()) {
    // partition helper function
    // find A[i]: the leftmost item >= pivot
    while (i < end && comp(vector.at(i), p_val)) {
        i++;
    }
    // find A[j]: the rightmost item < pivot
    while (begin < j && !comp(vector.at(j), p_val)) {
        j--;
    }
    if (i < j) {
        // smaller to left, bigger to right
        swap_val(vector, i, j);
        find_ij(vector, begin, end, p_val, i, j, comp);
    } else {
        // move pivot to correct position
```

```
        swap_val(vector, begin, j);
    }
}

template <typename T, typename Compare = std::less<T>>
size_t partition_in(std::vector<T> &vector, size_t begin, size_t end, size_t
pivotat, Compare comp = Compare()) {
    // partition
    T p_val = vector.at(pivotat);
    size_t i = begin + 1, j = (end - 1);
    find_ij(vector, begin, end, p_val, i, j, comp);
    return j;
}

template <typename T, typename Compare = std::less<T>>
void quick_sort_inplace_helper(std::vector<T> &vector, size_t begin, size_t end,
Compare comp = Compare())
{
    // first element as pivot
    size_t pivotat = begin;
    // base case
    if ((end - begin) < 2 || begin > end) {
        return;
    }
    // partition
    pivotat = partition_in(vector, begin, end, pivotat, comp);
    // sort each part
    quick_sort_inplace_helper(vector, begin, pivotat, comp);
    quick_sort_inplace_helper(vector, pivotat + 1, end, comp);
}

template <typename T, typename Compare = std::less<T>>
void quick_sort_inplace(std::vector<T> &vector, Compare comp = Compare())
{
    // TODO: implement
    size_t begin = 0, end = vector.size();
    quick_sort_inplace_helper(vector, begin, end, comp);
}
```

Features:

1. Left part is smaller then pivot, right part is bigger
2. Each part is unordered and does not remain original order

Master Theorem

Recurrence: $T(n) \leq aT\left(\frac{n}{b}\right) + O\left(n^d\right)$

Then:

$$T(n) = \begin{cases} O\left(n^d \log n\right) & \text{if } a = b^d \\ O\left(n^d\right) & \text{if } a < b^d \\ O\left(n^{\log_b a}\right) & \text{if } a > b^d \end{cases}$$

Non-comparison Sort

General concepts

Counting Sort

Sort an array `A` of integers in the range $[0, k]$, where k is known.

Procedure:

1. Allocate an array `count[k+1]`. Scan array `A`.
2. For $\mathrm{i} = 1$ to $\mathbf{N}$, increment `count[A[i]]`.
3. Scan array `count`. For $\mathrm{i} = 0$ to $\mathbf{k}$, print $\mathbf{i}$ for `count[i]` times.

Time complexity: $O(N + k)$**.**

The algorithm can be converted to sort integers in some other known range $[a, b]$: minus each number by a, converting the range to $[0, b - a]$.

To guarantee stability, instead of directly printing out, do:

1. For $i = 1$ to $\mathbf{k}$, `count[i]=count[i-1]+count[i]`. Then `count[i]` contains number of items less than or equal to i.
2. For $i = N$ **downto** 1, put `A[i]` in new position `count[A[i]]` and **decrement** `count[A[i]]`.

Bucket Sort

Algorithm:

1. Set up an array of initially empty "buckets".
2. **Scatter**: Go over the original array, putting each object in its bucket.
3. Sort each non-empty bucket by a **comparison sort**.
4. **Gather**: Visit the buckets in order and put all elements back into the original array.

Stability: Same as the comparison sort chosen.

Time complexity:

- Suppose we are sorting cN items and we divide the entire range into N buckets.
- Assume that the items are **uniformly distributed** in the entire range.
- The average case time complexity is $O(N)$, since **the comparison sort in each bucket takes** $O(c)$ **and** c **is constant**, and the initial assignment and the final readout take $O(cN)$ time.

Radix Sort

Sort integers by looking at **one digit at a time**.

Procedure:

- Given an array of integers, from the **least significant bit** (LSB) to the **most significant bit** (MSB), repeatedly do **stable** bucket sort according to the current bit.
- For sorting **base-**b numbers, bucket sort needs b **buckets**.

Time complexity:

- Let k be the maximum number of digits in the keys and N be the number of keys.
- We need to repeat bucket sort k times.
- Time complexity for the bucket sort is $O(N)$.

- **The total time complexity is $O(kN)$.**

Radix sort can be applied to sort keys that are built on **positional notation**.

Positional notation: all positions uses the same set of symbols, but different positions have different weight.

Linear Time Selection

Global assumption:

- array `A` with n **distinct** numbers, want to find i-th smallest element in the array.
- **index starts from 1**

Randomized Selection

```
Rselect(int A[], int n, int i) {
    // find i-th smallest item of array A of size n
    if(n == 1) return A[1];
    Choose pivot p from A uniformly at random;
    Partition A using pivot p;
    Let j be the index of p;
    // pivot is what we want
    if(j == i) return p;
    // find in the part smaller/bigger than pivot according to comparison
    if(j > i) return Rselect(1st part of A, j-1, i);
    else return Rselect(2nd part of A, n-j, i-j);
    }
```

Time complexity: for every input array of length n, the average runtime of `Rselect` is $O(n)$.

- Holds for every input data (**no assumption on data**)
- **"Average" is over random pivot** choices made by the algorithm

Good pivot

If `Rselect` chooses a pivot so that the left sub array's size is am, where $a \in [\frac{1}{4}, \frac{3}{4}]$ and m is the old length.

Best pivot: median

Deterministic selection algorithm

`ChoosePivot(A,n)` -- A subroutine called by the deterministic selection algorithm.

Procedure: (**median of medians**)

1. Break A into $n/5$ groups of **size** 5 each
2. Sort each group
3. Copy $n/5$ **medians** (the third element) into new array `C`
4. Recursively compute median of `C` by **calling the deterministic selection algorithm**
5. Return the median of `C` as pivot

```
Dselect(int A[], int n, int i) {
// find i-th smallest item of array A of size n
  if(n == 1) return A[1];
  Break A into groups of 5, sort each group;
  C = n/5 medians;
  p = Dselect(C, n/5, n/10);              ChoosePivot
  Partition A using pivot p;
  Let j be the index of p;
  if(j == i) return p;
  if(j > i) return Dselect(1st part of A, j-1, i);
  else return Dselect(2nd part of A, n-j, i-j);
}
```

Same as Rselect

Two recursive calls

Time complexity: For every input array of length n, `Dselect` runs in $O(n)$ time.

- not as good as `Rselect` in practice
- **Worse constants**
- **Not-in-place**: Need an additional array of $n/5$ medians

Hash!

Hashing basics

~~不想hash，再说吧~~ 还是得hash

Setup: A universe U of objects, want to maintain an evolving set $S \subseteq U$.

Solution:

- Pick an array `A` of n buckets. The array is called **hash table**.
  - $n = c|S|$: a small multiple of $|S|$.
- Choose a hash function $h : U \to \{0, 1, \ldots, n-1\}$, with following properties:
  - h is fast to compute.
  - The same key is always mapped to the **same** location.
- Store item k in $A[h(k)]$, $h(k)$ is called the **home bucket** of key k.

collision: **different** search keys go into the **same** bucket

Hash Function Design Criteria

Hash function $h(key) = c(t(key))$ maps key to buckets in two steps:

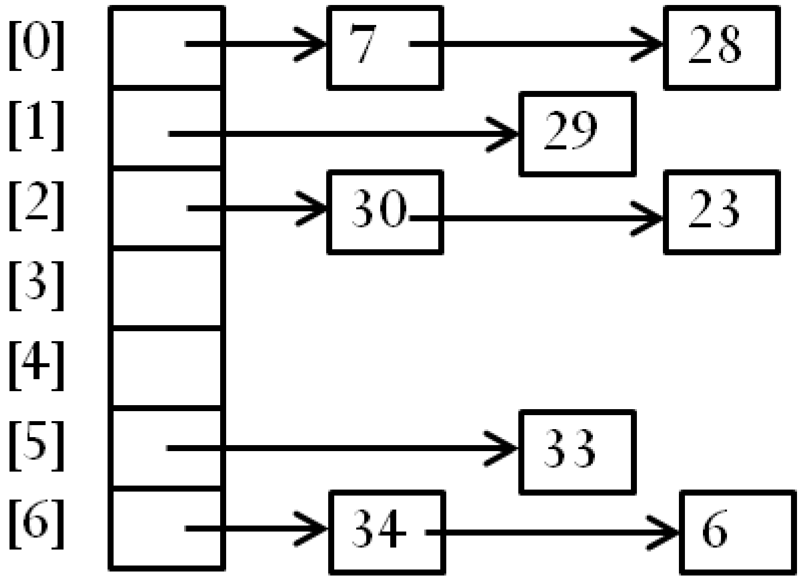
1. $t(key)$: Convert key into an integer, known as **hash code**, in case the key is not an integer.
2. $c(hashcode)$: **Compression map**: Map an integer (hash code) into a home bucket.
   - The most common method is by **modulo arithmetic**:

     $homeBucket = c(hashcode) = hashcode\%n$
     where n is the **number of buckets** in the hash table, a **large prime number**.

Collision Resolution

Separate chaining

Each bucket keeps a **linked list** of all items whose home buckets are that bucket.



`Value find(Key key)`

- Compute `k = h(key)`
- Search in the linked list located at the k-th bucket with the key.

`void insert(Key key, Value value)`

- Compute `k = h(key)`
- Search in the linked list located at the k-th bucket. If found, update its value; otherwise, insert the pair **at the beginning** of the linked list in $O(1)$ time.

`Value remove(Key key)`

- Compute `k = h(key)`
- Search in the linked list located at the k-th bucket. If found, remove that pair.

Open Addressing

Reuse empty space in the hash table to hold colliding items.

Linear Probing

Insert with: $h_i(key) = (h(key) + i)\%n$, where i is the number of collisions.

Find: probe in the buckets given by $h_0(key)$, $h_1(key)$, ..., in sequence until

- we **find the key**,
- or we find an **empty** slot, which means the key is **not found**.

Remove: mark deleted entry as "**deleted**", different from **empty**.

Clustering: when **contiguous** buckets are all occupied.

- Problems with a **large** cluster:
  - It becomes **more likely** that the next hash value will **collide** with the cluster.
  - **Collisions** in the cluster get **more expensive to resolve**.

Quadratic Probing

Insert with: $h_i(key) = \big(h(key) + i^2\big)\%n$.

Less likely to form large cluster.

Problem: sometimes we will never find an empty slot even if the table isn't full.

load factor :
$L = \frac{m}{n} = \frac{\#objectsinhashtable}{\#bucketsinhashtable}$

- if the **load factor** $L \leq 0.5$, we are guaranteed to find an empty slot (Quadratic Probing).
- open addressing requires $L \leq 1$.
- $L = O(1)$ is a necessary condition for operations to run in constant time.

Double Hashing

Insert with: $h_i(x) = (h(x) + i * g(x))\%n$ (use two **different hash functions**)

Rehashing

Determine Hash Table Size

1. $L \leq \frac{4}{5}$ to maintain $O(1)$ time complexity
2. choose a prime number

Rehash

Create a **larger** table, scan the current table, and then **insert** items into new table using the new hash function.

Note: The order is **from the beginning to the end** of the current table. Not original insertion order.

Amortized time complexity: $O(1)$

Application

De-Duplication, 2-sum problem

Universal Hashing

A **randomized** algorithm **H** for **constructing hash functions** $h : \mathrm{U} \rightarrow \{1, \ldots, \mathrm{M}\}$

H is universal if: for all $x \neq y$ in U,$\mathrm{Pr_h} \leftarrow \mathrm{H}[\mathrm{h(x)} = \mathrm{h(y)}] \leq 1/\mathrm{M}$

H is also called as a **universal hash function family**

Bloom Filter

Supports **fast insert** and **find**

Comparison to hash tables:

- Pros: more **space efficient**
- Cons:
    - Can't store an associated object

- **No deletion** (complicated deletion)
- Small **false positive** probability: may say x has been inserted even if it hasn't been. (But **no false negative** (x is inserted, but says not inserted).)

Implementation

An array of n bits. Each bit **0 or 1**

- $n = b|S|$, where b is small real number. For example, $b \approx 8$ for 32-bit IP address (That's why it is space efficient)

k hash functions h_1,..., h_k, each mapping inside $\{0, 1, \ldots, n-1\}$.

- k usually small.

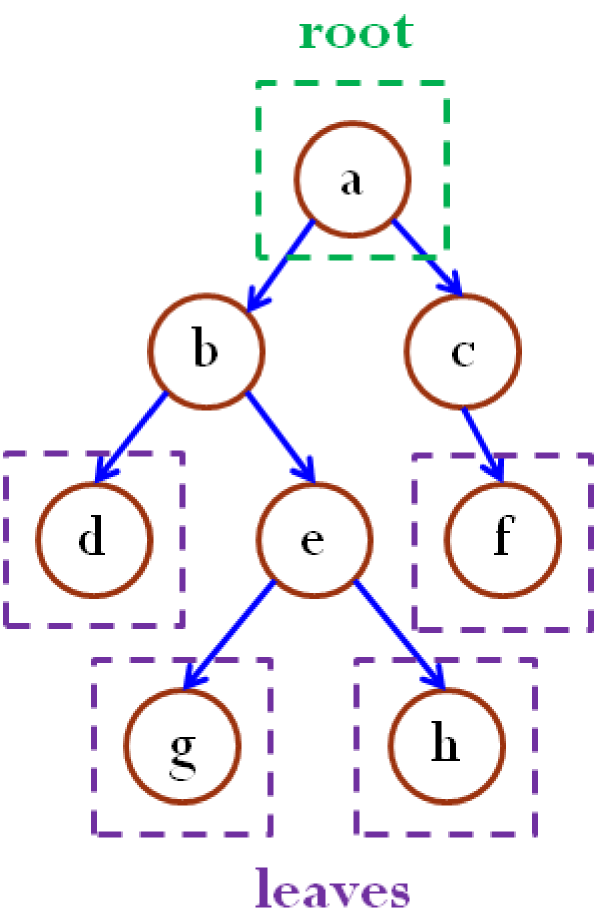
Insert x : For $i = 1, 2, \ldots, k$, set $A[h_i(x)] = 1$

Find x : return true if and only if $A[h_i(x)] = 1, \forall i = 1, \ldots, k$

Trees

Concepts

An extension of linked list data structure: Each node connects to **multiple** nodes.



- The node **at the top** of the hierarchy is the **root**.
- Nodes are connected by **edges**.
- Edges define **parent-child** relationship.
- Root has no parent.
- All other nodes have **exactly one** parent.

- A node with **no children** is called a **leaf**.
- Nodes that share **the same parent** are **siblings**.

Subtree of a node `r` is a tree rooted at **a child of node** `r`.

Path

A **path** is a sequence of nodes such that the next node in the sequence is a child of the previous.

Path length: the **number of edges** in the path

- Path length may be 0
- Claim: If there exists a path between two nodes, then this path is the **unique** path between these two nodes. (Because a node can only have one parent)
- If there exists a path from a node A to a node B, then A is an **ancestor** of B and B is a **descendant** of A.

Depth, Level, Height

Depth/Level: the length of the unique path from the **root** to the node.

- E.g., depth(b)=1, depth(a)=0.

Height: the length of the **longest** path from the node to a **leaf**.

- E.g., height(b)=2, height(a)=3.
- All **leaves** have height **zero**.
- The **height of a tree** is the **height of its root**. == the depth of a tree
- The **number of levels of a tree** is the height of the tree **plus one**.

Degree

Degree: the number of children of a node.

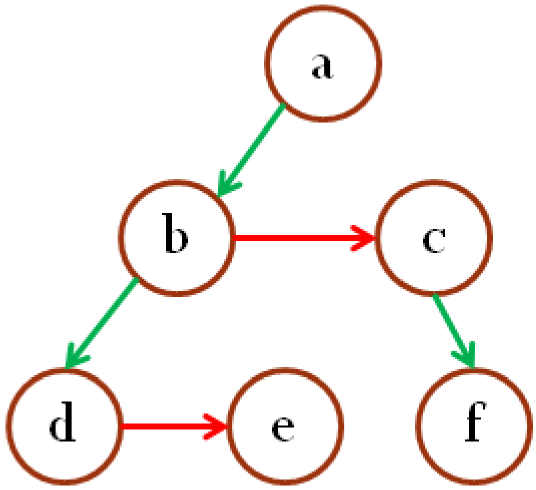
The **degree of a tree** is the **maximum** degree of a node in the tree.

Simple implementation of tree

Each node is part of a **linked list** of **siblings**.

Additionally, each node stores a pointer to its **first child**.

```
struct node {
    Item item;
    node *firstChild;
    node *nextSibling;
};
```



Binary Tree

Every node can only have **at most two** children.

(Including the empty tree and a single-node tree)

Properties

The **minimum number of nodes** in a binary tree of height h is: $h+1$. (One node one level)

The **maximum number of nodes** in a binary tree of height h is: $2^{h+1}-1$.

- At most 2^k nodes at level k.
- $1+2+2^2+\cdots+2^h=2^{h+1}-1$

Let n be the number of nodes in a binary tree whose height is h:

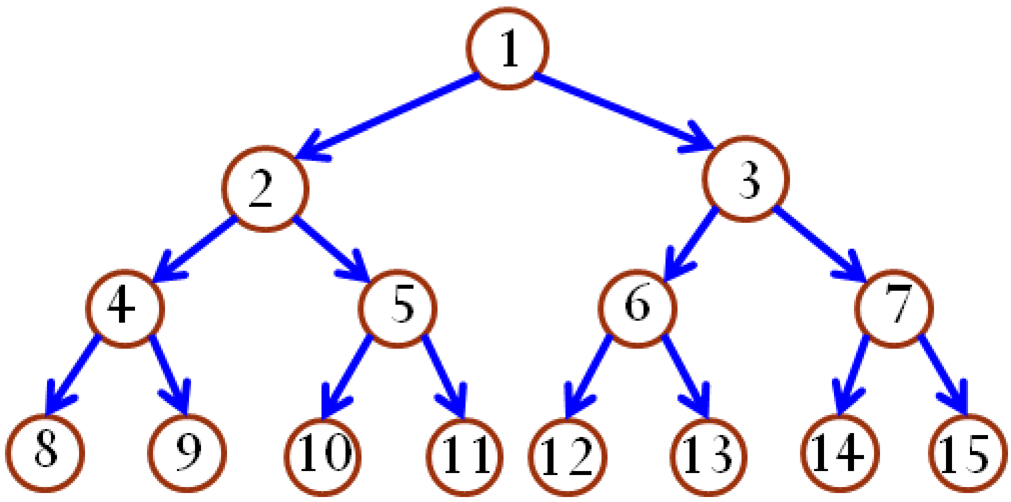
- $h+1 \le n \le 2^{h+1}-1$
- $\log_2(n+1)-1 \le h \le n-1$

Types

1. **proper**: every node has **0 or 2 children**.
2. **complete**:
    1. every level **except the lowest** is fully populated, and
    2. the lowest level is populated **from left to right**.
3. **perfect**: every level is fully populated

Numbering Nodes In a Perfect Binary Tree

- Numbering **from top to bottom** level.
- Within a level, numbering **from left to right**.



- What is the **parent** of node i?
  - For $i \neq 1$, it is $\lfloor i/2 \rfloor$.
- What is the **left child** of node i?
  - If $2i \leq n$, it is $2i$; If $2i > n$, no left child.
- What is the **right child** of node i?
  - If $2i+1 \leq n$, it is $2i+1$; If $2i+1 > n$, no right child.

We can use this to represent binary trees using arrays.

Representing Binary Tree Using Linked Structure

```
struct node {
    Item item;
    node *left;
    node *right;
};
```

Binary Tree Traversal

DFS

Pre-Order

node->left subtree->right subtree

```
void preOrder(node *n) {
    if(!n) return;
    visit(n);
    preOrder(n->left);
    preOrder(n->right);
  }
```

Post-Order

left subtree->right subtree->node

```
void postOrder(node *n) {
    if(!n) return;
    postOrder(n->left);
    postOrder(n->right);
    visit(n);
}
```

In-Order

left subtree->node->right subtree

```
void inOrder(node *n) {
    if(!n) return;
    inOrder(n->left);
    visit(n);
    inOrder(n->right);
}
```

BFS

Level-Order Traversal

from **top to bottom**, from **left to right**

implement: use a queue (first in first out)

1. **Enqueue the root** node into an empty queue.
2. While the queue is **not empty**, **dequeue** a node from the front of the queue.
    - **Visit** the node.
    - **Enqueue its left child** (if exists) and **right child** (if exists) into the queue.

```
void levelOrder(node *root) {
    queue q; // Empty queue
    q.enqueue(root);
    while(!q.isEmpty()) {
        node *n = q.dequeue();
        visit(n)
        if(n->left) q.enqueue(n->left);
        if(n->right) q.enqueue(n->right);
    }
}
```

Application

- The expression $a/b + (c–d)e$ has been encoded as a tree `T`.
    - The leaves are **operands**.
    - The internal nodes are **operators**.

- How would you traverse the tree `T` to print out the expression (ignoring parentheses)?
  - **In-order** depth-first traversal.
- What is the expression printed out by post-order depth-first traversal?
  - $ab/cd - e * +$
  - **Reverse Polish Notation**

Priority Queues and Heaps

Priority Queue

Min(Max) Priority Queue

Min Priority Queue: A collection of items and each item has a key (or “**priority**”).

Support the following operations:

- `isEmpty`
- `size`
- `enqueue`: put an item into the priority queue.
- `dequeueMin`: remove element with **min** key.
- `getMin`: get item with **min** key.

unsorted array-based implementation:

`isEmpty` / `size` / `enqueue`: $O(1)$; `dequeueMin`: $O(n)$; `getMin`: $O(n)$

Complexity of the operation using heap implementation:

- `isEmpty`, `size`, and `getMin` are $O(1)$ time complexity in the worst case.
- `enqueue` and `dequeueMin` are $O(logn)$ time complexity in the worst case, where n is the size of the priority queue.

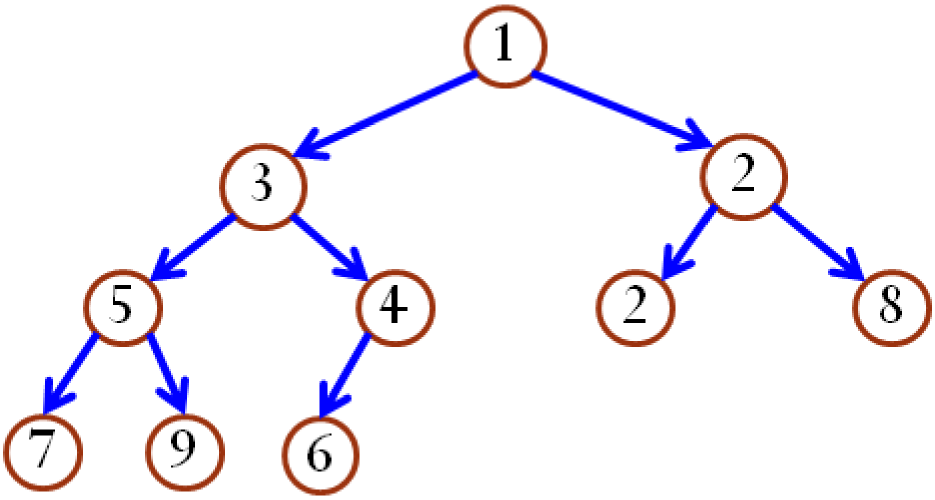
Binary Heap

A **binary heap** is a **complete binary tree**.

Min Heap

- a **binary heap**, and
- a tree where for **any** node `v`, the key of `v` is $\leq$ the keys of any **descendants** of `v`.

Property: The key of the root of **any** subtree is always the **smallest** among all the keys in that subtree.



Index Relation

- A node at index $i(i \neq 1)$ has its parent at index $\lfloor i/2 \rfloor$.
- Assume the number of nodes is n. A node at index $i(2i \leq n)$ has its left child at $2i$. If $2i > n$, it has no left child.
- A node at index $i(2i+1 \leq n)$ has its right child at $2i+1$. If $2i+1 > n$, it has no right child.

Min Heap Implementation

```
isEmpty: return size==0;
size: return size;
getMin: return heap[1];
```

`enqueue` $O(\log n)$

1. Insert `newItem` as the **rightmost leaf** of the tree.
2. `PercolateUp(newItem)` to an appropriate spot in the heap to restore the heap property.

```
void minHeap::percolateUp(int id) {
  while(id > 1 && heap[id/2] > heap[id]) {
    swap(heap[id], heap[id/2]);
    id = id/2;
  }
}
```

- Pass index (**id**) of array element that needs to be percolated up.
- **Swap** the given node **with its parent** and move up to parent until:
  - we **reach the root** at position 1, or
  - the **parent has a smaller or equal key**.

```
void minHeap::enqueue(Item newItem) {
  heap[++size] = newItem;
  percolateUp(size);
}
```

`PercolateUp(newItem)` is also used when `DecreseKey`

`dequeueMin` $O(\log n)$

1. The min item is at the **root**. **Save** that item to be **returned**.
2. Move the item in the **rightmost leaf** of the tree **to the root**.
3. `PercolateDown(newRoot)`

```
void minHeap::percolateDown(int id) {
  for(j = 2*id; j <= size; j = 2*id) {
    if(j < size && heap[j] > heap[j+1]) j++; // find smaller child
    if(heap[id] <= heap[j]) break;
    swap(heap[id], heap[j]);
    id = j;
  }
}

Item minHeap::dequeueMin() {
  swap(heap[1], heap[size--]);
  percolateDown(1);
  return heap[size+1];
}
```

Initializing Min Heap

- Put all the items into a complete binary tree.
- Starting at the rightmost array position **that has a child**, percolate down all nodes in **reverse** level-order. (right to left, bottom to up)
  - The rightmost array position **that has a child** is **size/2**. (array notation)

Time complexity: $O(n)$

Heap sort

1. Initialize a min heap
2. Repeatedly call `dequeueMin`

Time complexity: $O(n \log n)$

Median Maintenance

Input: a sequence of numbers $x_1, x_2, \ldots, x_n$, one-by-one

Output: at each time step i, the median of $x_1, x_2, \ldots, x_i$

using two heaps, one **min heap** and **one max heap**

Key idea: maintain the **smallest** half $\left(\lceil \frac{n}{2} \rceil\right)$ in **max heap** and the largest half $\left(\lfloor \frac{n}{2} \rfloor\right)$) in the min heap

`median = maxHeap.getMax()`

Binary Search Tree

A **binary search tree (BST)** is a binary tree with the following properties:

- Each node is associated with a **key**.
  - A key is a value that can be compared.

    - Assume: all the keys are **distinct**.
- The key of **any** node is **greater** than the keys of all nodes in its **left subtree** and **smaller** than the keys of all nodes in its **right tree**.

```
struct Item {
    Key key;
    Val val;
};
struct node {
    Item item;
    node *left;
    node *right;
};
```

A BST allows `search`, `insertion`, and `removal` by key.

The **average case** time complexities for these operations are $O(\log n)$.

- Average over all possible BSTs.

`search`

```
node *search(node *ptr, Key k) {
    if(ptr == NULL) return NULL;
    // find
    if(k == ptr->item.key) return ptr;
    // compare, and go left or right
    if(k < ptr->item.key)
        return search(ptr->left, k);
    else return search(ptr->right, k);
}
```

`insert`

```
void insert(node *&root, Item item) {
    // EFFECTS: insert the item as a leaf,
    // maintaining the BST property.
    if(root == NULL) {
        root = new node(item);
        return;
    }
    // insert as leaf
    if(item.key < root->item.key)
        insert(root->left, item);
    else if(item.key > root->item.key)
        insert(root->right, item);
}
```

`remove`

```
void remove(node *&root, Key k) {
    // base case
    if(root == NULL) return;
    // recursion
```

```
    if(k < root->item.key) remove(root->left, k);
    else if(k > root->item.key)
        remove(root->right, k);
    else { // root->item.key == k
        if(isLeaf(root)) { // remove leaf
            delete root;
            root = NULL;
        } else { // remove degree-one or two node
            if(root->right == NULL) { // no right child
                node *tmp = root;
                root = root->left;
                delete tmp;
            } else if(root->left == NULL) { // no left child
                node *tmp = root;
                root = root->right;
                delete tmp;
            } else { // remove degree-two node
                // Replace with the largest key in the left subtree.
                node *&replace = findMax(root->left);
                root->item = replace->item;
                node *tmp = replace;
                replace = replace->left;
                // both leaf and degree-one node are OK
                delete tmp;
            }
        }
    }
}

node *&findMax(node *&root) {
  if(root->right == NULL) return root;
  return findMax(root->right);
}
```

- Node to be removed is a leaf.
    - Delete the node.
- Node to be removed is a degree-one node.
    - “Bypass” the node from its parent to its child.
- Node to be removed is a degree-two node.
    - Replace the node key with the largest key in the left subtree and remove the node with the largest key

Average Case Time Complexity of BST

$O(\log n)$ for a successful search.

没事别看数学证明 会变得不幸