Data Structures

Author: Jose 胡冠洲 @ ShanghaiTech

Arrays

Simple Array

Definition

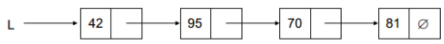
Performance

- Access k-th Entry: $\Theta(1)$
- Insert or Erase at
  - Front: $\Theta(n)$
  - k-th: $O(n)$
  - Back: $\Theta(1)$

Linked Lists

Node-based Impl.

Definition



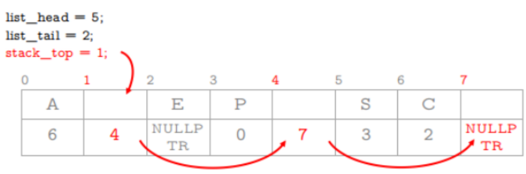
Performance

	Front/1^{st} node	k^{th} node	Back/n^{th} node
Find	$\Theta(1)$	$O(n)$	$\Theta(1)$
Insert Before	$\Theta(1)$	$\Theta(1)^*$	$\Theta(1)$
Insert After	$\Theta(1)$	$\Theta(1)^*$	$\Theta(1)$
Replace	$\Theta(1)$	$\Theta(1)^*$	$\Theta(1)$
Erase	$\Theta(1)$	$\Theta(1)^*$	$\Theta(n)$
Next	$\Theta(1)$	$\Theta(1)^*$	n/a
Previous	n/a	$O(n)$	$\Theta(n)$

> Achieved with help of `list_tail` pointer.

Array-based Impl.

Definition

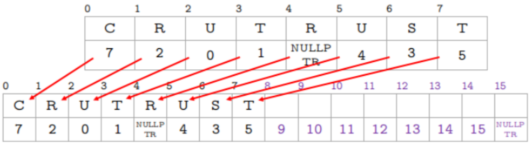


- `list_head` points to first index
  - Every cell points to next index
  - Tail cell contains `NULL`
- `stack_top` points to first empty index
  - Every empty cell points to next empty index
  - Last empty cell contains `NULL`

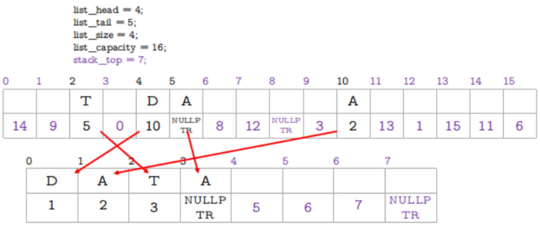
Operations

- *Pushing* & *Poping*
  - Insert into `stack_top` / Push empty cell into stack

- Remember to modify `stack_top` & `list_head`!
- *Reallocation*
  - *Enlarge*:



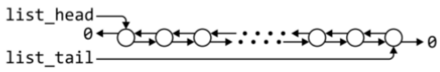
    - Remember to **update** `capacity` & `stack_top`
  - *Shrink*:



    - Remember to **update all members**

Doubly-linked List

Definition



Performance

	Front/1st node	k^{th} node	Back/n^{th} node
Find	$\Theta(1)$	$O(n)$	$\Theta(1)$
Insert Before	$\Theta(1)$	$\Theta(1)^*$	$\Theta(1)$
Insert After	$\Theta(1)$	$\Theta(1)^*$	$\Theta(1)$
Replace	$\Theta(1)$	$\Theta(1)^*$	$\Theta(1)$
Erase	$\Theta(1)$	$\Theta(1)^*$	$\Theta(1)$
Next	$\Theta(1)$	$\Theta(1)^*$	n/a
Previous	n/a	$\Theta(1)^*$	$\Theta(1)$

Stacks

Last in, First out (LIFO).

- Two principal operations, both $\Theta(1)$
  - `push` to top
  - `pop` the top

Singly-linked List Impl.

Definition

```
template <typename Type>
class Stack {
    private:
        Single_list<Type> list;
    public:
        bool empty() const;
        Type top() const;
        void push( Type const & );
        Type pop();
};
```

One-ended Array Impl.

Operations

- *Enlarging* Schemes
  - $+= 1$ every time
    - Each push $\Theta(n)$ time

        - Wasted space $\Theta(1)$
    - $* = 2$ every time
        - Each push $\Theta(1)$ time
        - Wasted space $\Theta(n)$

> Applications of Stacks
>
> - XML matching
> - C++ Parsing
> - Function calls
> - Post-fix (Reverse-Polish) notation

Queues

First in, First out (FIFO).

- Two principal operations, both $\Theta(1)$
    - `enqueue` to bottom
    - `dequeue` the top

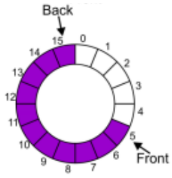
Singly-linked List Impl.

Definition

```
template <typename Type>
class Queue{
    private:
        Single_list<Type> list;
    public:
        bool empty() const;
        Type front() const;
        void enqueue( Type const & );
        Type dequeue();
};
```

Circular Array Impl.

Definition

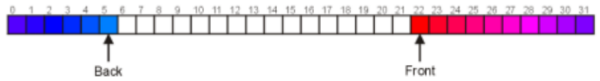



```
template <typename Type>
class Queue{
    private:
        int queue_size;
        int ifront; // Initially 0.
        int iback;  // Initially -1.
        int array_capacity;
        Type *array;
    public:
        Queue( int = 10 );
        ~Queue();
        bool empty() const;
        Type front() const;
        void enqueue( Type const & );
        Type dequeue();
};
```

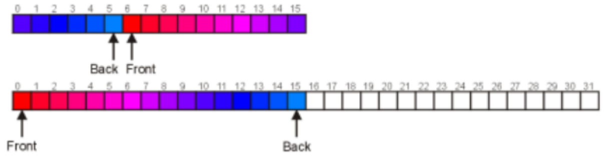
Operations

- *Enlarging* Schemes

- Solution 1:

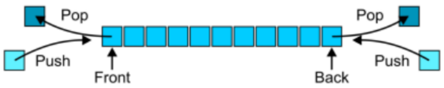


- Solution 2:



Double-ended Queue (Deque)

Definition



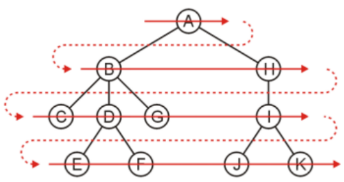
Trees: Introduction

Terms & Properties

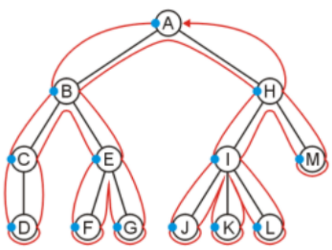
- Terms
  - *Root*, *Leaf*, *Internal Nodes* (including Root)...
  - *Path*, *Depth*(length of path from root)...
  - ***Height***: maximum depth, **Count # of edges, NOT nodes**
    - Only Root $\Rightarrow 0$
    - Empty $\Rightarrow -1$
  - *Ancestor*, *Descendant* (including the Node itself)
- Properties
  - Recursive definition: Subtrees
  - Each Node, other than Root, has exactly one node pointing to it
  - No Loops, n nodes $\rightarrow n-1$ edges
  - Detach first before Attaching

Tree Traversal

- **BFS** (*Breadth-First Traversal*): use **Queue**, $\Theta(n)$

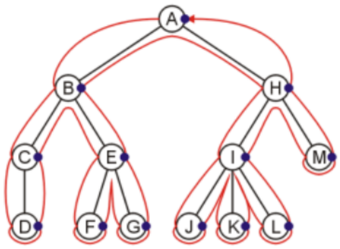


- **DFS** (*Depth-First Traversal*): use **Recursion / Stack**, $\Theta(n)$
  - **Pre-order**, mark when first visited



A, B, C, D, E, F, G, H, I, J, K, L, M

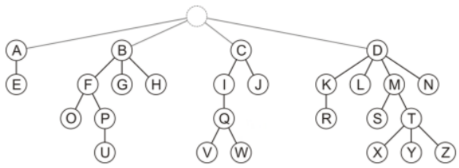
- **Post-order**, mark when leaving it



D, C, F, G, E, B, J, K, L, I, M, H, A

Forest

Definition

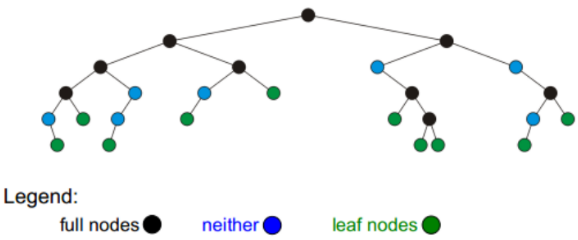


- Collection of Disjoint Rooted Trees

> Traversal can be achieved by treating the roots as children of a *Notional Root*.

Trees: Binary Tree

Definition



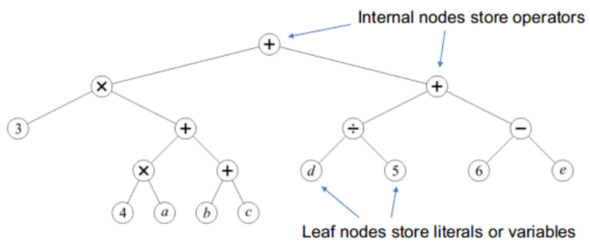
- Notations
  - *Full* binary tree: each node is full / leaf, $\neq$ *Complete*
  - *Complete*: All left-most nodes are filled
  - *Perfect*: All leaf at same depth; all other nodes are full
    - $2^{h+1} - 1$ nodes
    - Height $h = \lg(n+1) - 1$

```
template <typename Type>
class Binary_node {
    protected:
        Type element;
        Binary_node *left_tree;
        Binary_node *right_tree;

    public:
        Binary_node( Type const & );

        Type retrieve() const;
        Binary_node *left() const;
        Binary_node *right() const;

        bool is_leaf() const;
        int size() const;
};
```

Operations

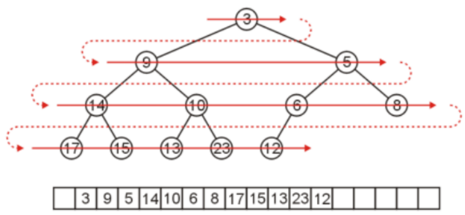
- Traversals
  - Pre-order (先根)
  - In-order (中根)
  - Post-order (后根)
- NOTE: How many different forms a binary tree with height h can have? A: Catalan #: $\binom{2n}{n} - \binom{2n}{n-1}$

Expression Tree



- Use Post-ordering DFS to get Reverse-Polish format
- Use In-order Traversal to get Infix format
  - Need to add *Brackets*!!!
  - 符号 prev-$\longrightarrow$ 符号, add "("
  - 符号 $\longleftarrow$-back 符号, add ")"

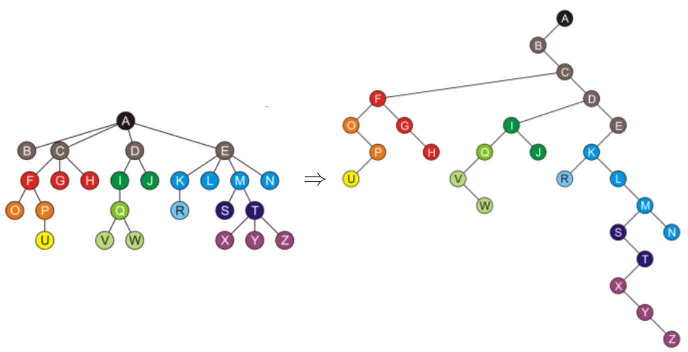
Complete Binary Tree




```
parent = k >> 1;
left_child = k << 1;
right_child = left_child + 1;
```

Left-child Right-sibling Binary Tree

- *Knuth Transform*: Store general tree as Binary Trees.
  - Empty left sub-tree $\Rightarrow$ no children
  - Empty right sub-tree $\Rightarrow$ last in its siblings
  - Pre-order traversal identical
  - Post-order traversal of original tree $=$ In-order traversal of Binary



> For forests, consider all roots to be siblings.

Trees: Binary Heaps

First in, Highest priority out (A specific implementation of Priority Queues).

Definition

Take a Min-heap for example:

- Key of Root $\leq$ Keys of subtrees
- Subtrees are also Min-heaps
- Usually, use **Complete** Binary Trees to *ensure Balanceness* $\Rightarrow$ Space: $O(n)$

Operations

Pop Root:

1. Remove root, **replace with the last node**
2. From the new root, (*percolate*)
    1. **If smaller than both children, DONE**
    2. **Else, Swap with smaller child**
    3. Go down and **Recurse**

Push (Insert):

1. Add to the first empty slot
2. From the inserted node, (*percolate*)
    1. **If bigger than parent, DONE**
    2. **Else, Swap with parent**
    3. Go up and **Recurse**

Build Heap - Floyd's Method:

1. for $i = \frac{size}{2}$ to 0
    1. Percolate down `array[i]`

Observations on this method:

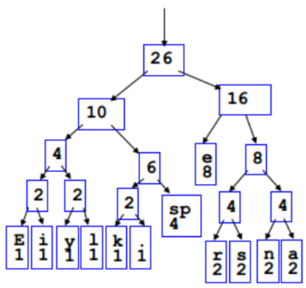
- We can directly use an Array to represesnt a Heap
- For a Complete Tree, $\lceil \frac{size}{2} \rceil$ Leaf Nodes
- Only those **Non-leaf Nodes need** *percolation* $\Rightarrow$ Time complexity is $\Theta(n)$

Heapsort

1. Use Floyd's Method to build a *max-heap* as array
2. Pop the root
    - This will make an empty space near end of array
    - Put the poped element there
    - **Repeat** until finish

Huffman Coding

Char	**Code**
E	0000
i	0001
y	0010
l	0011
k	0100
.	0101
space	011
e	10
r	1100
s	1101
n	1110
a	1111

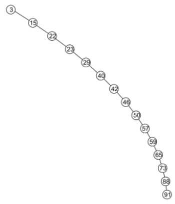


1. Scan text, count frequencies
2. Build *Huffman-Tree*
    1. Pick smallest two
    2. Combine
    3. Push back
3. Traverse through *Huffman-Tree* to determine code
    - Left gets 0, Right gets 1
4. Go through text to encode

Trees: Binary Search Trees (BST)

Definition

- Left sub-tree (if any) is a BST and all Elements are less than the Root
- Right sub-tree (if any) is a BST and all Elements are larger than the Root

> Worst case of a BST: 

Operations

Find Minimum (Maximum): ($O(h)$)

1. Go to left (right) most node

Find: ($O(h)$)

1. Do Binary Search
2. If empty node reached, return NULL

Insert: ($O(h)$)

1. Do `find`
2. If found, return NULL
3. Else insert at that empty node

Find Successor:

1. **If has a right subtree**, do `find_minimum` in right subtree
2. **Else**, go up toward root
    1. Find the first larger object on this path

Delete:

1. **Case 1**: leaf, delete directly
2. **Case 2**: has one child, replace by this child
3. **Case 3**: has two children
    1. Find the **Successor**
    2. Replace by the successor
    3. **Delete the successor**

Find k-th Object

1. **If** `size(left_subtree)` **==** k, return current node
2. **If** `size(left_subtree)` **>** k, go to left subtree, and **Recurse**
3. **Else**, go to right subtree
    1. **Recurse** on finding the ($k-$`size(left_subtree)`-1)-th entry

Trees: AVL Tree

Definition

AVL Balanced means:

- $|h(l) - h(r)| \leq 1$
- Both left subtree and right subtree are balanced

Must-Know Patterns

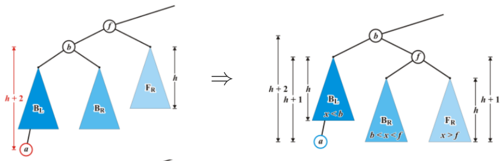
- # of nodes upper bound $= 2^{h+1} - 1$
- # of nodes lower bound

- $F(h) = F(h-1) + 1 + F(h-2)$
- $F(h) = Fibonacci[h+3] - 1$

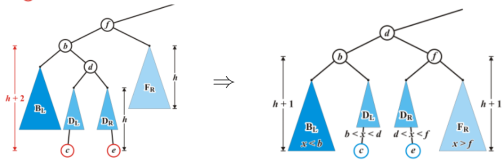
Operations

Insertion:

1. Follow Binary Tree Convention
2. Check unbalanceness, Find **the LOWEST unbalanced node**:
    - **L-L** (symmetric to **R-R**)



    - **L-R** (symmetric to **R-L**)



Deletion

1. Follow Binary Tree Convention
2. **Trace back to root, Check unbalanceness**!
    - Rotate to fix, then go upward
    - Need to check every node on this path! (include Root)

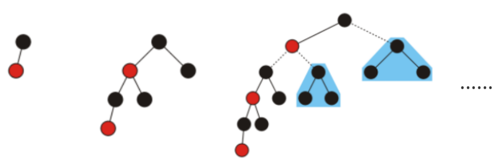
Trees: Red-Black Tree (RBT)

Definition

- Each node Red / Black (1 bit)
- *Null Path*: path starting from the root where the last node is not full
- **Restrictions**:
    1. **Root must be black**
    2. **Red node can only have black children**
    3. **Each null path have same # of black nodes**

Must-Know Patterns

- **Red node** must be either **full** / **leaf**
- If node *P* has exactly one child *Q*:
    - *Q* must be red
    - *Q* must be leaf
    - *P* must be black
- Worst case RB Trees
    - k: # of black nodes per null path, h: height
    - $F(k)$ (total # of nodes): $F(k) = F(k-1) + 2 + 2(2^{k-1} - 1)$
    - $h_{worst} = 2\lg(n+2) - 3$

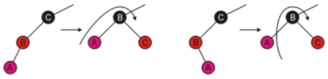


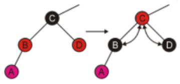
Operations

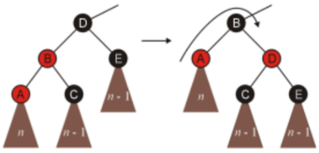
Bottom-Up Insertion:

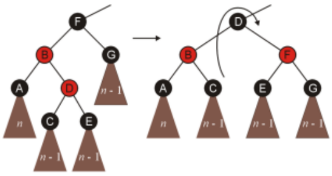
1. **MUST insert a red node**, o.w. The global rule c. will be violated
2. Find the place where it will be inserted

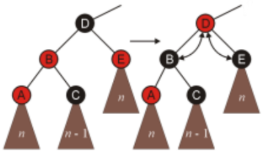
3. **If parent is black, DONE**
4. **Else if parent is Red**

    1. **If grandparent has only one child:**  Then **DONE**

    2. **If grandparent has two children:** (uncle must be red)  Then ***Recurse***!

        1. **Recursion Case 1**:  Then **DONE**

        2. **Recursion Case 2**  Then **DONE**

        3. **Recursion Case 3**  Then **Continue Recursion**!

    3. **If the root becomes Red at last, Colour it Black**

Top-Down Insertion:

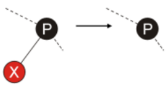
1. From the root, every step downward:
    1. Check: **current node is black AND there are two red children ?**
        1. **If true** ⇒ **Swap color**
        2. **If the root becomes Red, Colour it Black**
    2. Check: **After Swaping, parent AND self both Red ?**
        1. **If true, do a Rotation!**
2. When already reaches bottom, only needs at most one more rotation

Deletion:

When deleting a node:

- If Node is Leaf, delete it
- If Node is internal, replace it with its Successor, then **actually deletes the Successor**

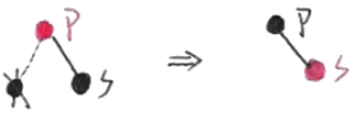
Therefore patterns can ONLY be:

1. **Deleting a Red leaf, DONE** 

2. **Deleting Black node with a Red leaf**, Replace value  Then **DONE**

3. **Deleting a Black leaf**! (不是最简但最直观)

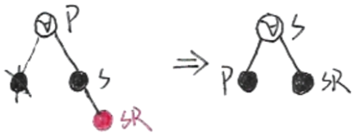
    1. **Sibling Red**: 

    2. **Sibling Black & Parent Red**: 

    3. **Sibling Black & Parent Black**: 

        - Now the whole subtree (rooted at *P*) lacks a black node, **Recurse**!

4. **Sibling Black & has left child**: 

5. **Sibling Black & has only right child**: 

Hash Tables

Scenary: Error code vary in range 0~65536, but in total 150 different error conditions

- Solution 1 - Array of length 150: Slow to locate an error code (Binary search)
- Solution 2 - Array of length 65536: Large memory space wasted
- Solution 3 - Hash Table

Mapping objects onto numbers

- *Predetermined*. e.g. Student ID #
  - May make two equivalent objects having different hash values
- *Arithmetic*, e.g. a determinstic function
  - For rational #s: define $\frac{p}{q} \Rightarrow p + q$
    - Problem 1 - $\frac{1}{2}$ and $\frac{2}{4}$ hashes differently: Divide by gcd
    - Problem 2 - $\frac{1}{2}$ and $\frac{-1}{-2}$ hashes differently: Use abs form
  - For strings: Let individual characters represent coefficients of polynomial of x

Hash Functions

The process of **mapping** a number **onto an integer index in a given range**.

- Requirements
  - Must be in $O(1)$ time
  - Output is **determinstic: Always same output for same input**
  - Could have *collision* situations
- Types
  - Modulus: $H(n) = n\%M =$ `n & ((1 << m) - 1)`, $M = 2^m$
    - Fast, just take last m bits
  - Multiplicative: $H(n) = \text{certain } m \text{ bits in } n * C =$ `((C * n) >> shift) & ((1 << m) - 1)`
  - ...

Collisions Dealing: Chained List

Use **linked lists** to store collisions.

- Operations
  - `push_front` to list head every time
  - Search should now go through the list ($O(\lambda)$)
- **Load Factor** $\lambda = \frac{n}{M}$ represents the length of linked lists
  - If λ goes high, **re-hash** (Double the table size and re-insert all elements)
  - Choose hash functions that avoid *clustering*!
- Could replace each linked list with an AVL tree

Collisions Dealing: Open Addressing - Linear Probing

Operations

Insert:

1. If bin k **empty, occupy it**
2. Otherwise, go to bin $k+1$, $k+2$... **until an empty bin is found** (*Circular array*!)

Search:

1. Start from bin k, **search forward until**
   - item found, `true`
   - empty bin found, `false`
   - traversed entire array, `false`

Erase:

1. **Erase slot** k, making a *hole* at bin k (`hole = k`)
2. Repeat:
   1. Go to next adjacent bin k' (occupied by element n)
   2. If $H(n)$ **is *at* OR *before*** `hole` **but after** k'
      - Move n into `hole`, resulting in a new *hole* (`hole = k'`)
3. Until next bin is empty

Lazy Erasing:

- Erase slot k, **mark it as** `ERASED`
- When **searching meets** `ERASED`**, regard it as occupied**
- When **insertion meets** `ERASED`**, regard it as un-occupied**
  - **Need searching before insertion** to avoid duplicate elements
- When calculating λ, regard `ERASED` as occupied

Analysis

- Primary Clustering: Probability of increasing length of a length-l cluster $= \frac{l+2}{M}$
- # of probes for a successful search: $\frac{1}{2}(1 + \frac{1}{1-\lambda})$
- # of probes for a un-successful search: $\frac{1}{2}(1 + \frac{1}{(1-\lambda)^2})$
- Keep λ under a certain *bound*

Collisions Dealing: Open Addressing - Quadratic Probing

Move forward by different amounts every time.

- Using $M = 2^m$, step forward 1 at first probing, then 2, 3...
- Must use *Lazy Erasing*
- Analysis
  - Secondary Clustering: Object hashing to same bin follows same sequence
  - # of probes for a successful search: $\frac{\ln \frac{1}{1-\lambda}}{\lambda}$
  - # of probes for a un-successful search: $\frac{1}{1-\lambda}$

Disjoint Sets (Union-Find Set)

- **Partition** elements according to *equivalence relations*
- Use a **representative** to represent all elements in set
  - `find(a)` operation returns the representative of set that `a` is in
  - `union(a, b)` operation unions two sets containing `a, b`

Array-based impl.

Definition

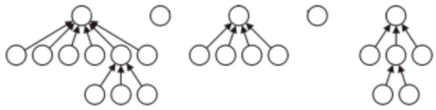
1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25
1	2	1	4	1	2	1	8	1	2	1	4	1	2	1	16	1	2	1	4	1	2	1	8	1

Performance

- `find` takes $\Theta(1)$
- `union` takes $\Theta(n)$

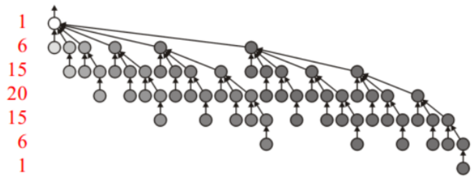
Tree-based impl.

Definition



```
size_t find( size_t i ) const {
    if (parent[i] != i)
        parent[i] = find(parent[i]);
    return parent[i];
}

void union(size_t i, size_t j) {
    i = find(i);
    j = find(j);
    if ( i != j )
        parent[j] = i;
}
```

Performance

- `find` takes $\Theta(h)$
  - Apply **path compression**
- `union` takes $\Theta(h)$
  - Point root of shorter tree to taller one
- Worst case: *Pascal's Triangle* 

  - Depth is $\frac{\ln n}{2} = O(\ln n)$ without path compression
    - Depth is $O(\alpha(n))$ with path compression

> Application of Disjoint Sets: *Maze Generation*
>
> 1. Divide the maze into square cells, each surrounded by four walls
> 2. Make every cell a disjoint set
> 3. Repeat:
>   1. Randomly choose a wall
>   2. If it connects two disjoint sets of cells, remove it, union two sets
> 4. Until all cells in one set

Graphs: Introduction

$G = (V, E)$, where V is set of *Vertices*, and E is set of *Edges*.

Categories

Undirected Graph

- **Vertex** & **Edge**
  - Assume $\{v_1, v_1\}$ (self-loop) is not an edge
  - $|E|_{max} = \frac{|V|(|V|-1)}{2} = O(|V|^2)$
- **Neighbours**: adjacent vertices
- **Degree** of a vertex: # of neighbours
- **Subgraph** $G_s = (V_s, E_s)$ where V_s is subset of V, E_s is subset of E
  - Vertex-included sub-graph $G' = (V, E_s)$
- **Path** from v_0 to v_k
  - *Length* of a path is # of edges it passes through
  - **Simple Path** has no repetition vertices (except maybe $v_0 = v_k$)

- **Simple Cycle** is a simple path with $v_0 = v_k$
- **Connected**: exists a path between
  - A **Connected Graph** has a path between any pair of vertices
  - A **Connected Component** is a maximum connected subgraph
- **Weight** of an edge might be assigned
  - *Length* of a weighted path is Sum of edge weights it passes through
- A **Tree** is:
  1. Connected & There is a unique path between any two vertices
  2. Have exactly $n - 1$ edges for n nodes
  3. *Acyclic*
  4. Removing any edge creates two unconnected sub-graphs
- A **Forest** is any graph that is *Acyclic*
  - \# of trees in forest $= |V| - |E|$

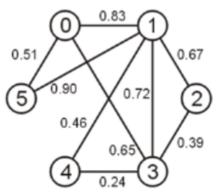
Directed Graph

- $|E|_{max} = |V|(|V| - 1) = O(|V|^2)$
- **In Degree** of a vertex: # of inward edges
  - In degree $= 0$ - *Source*
- **Out Degree** of a vertex: # of outward edges
  - Out degree $= 0$ - *Sink*
- **Connected**: exists a path between
  - **Strongly Connected**: there exists a directed path *from and to* any two vertices
  - **Weakly Connected**: view it as *undirected* and then connected
- *Directed Acyclic Graphs* (**DAG**)

Representations

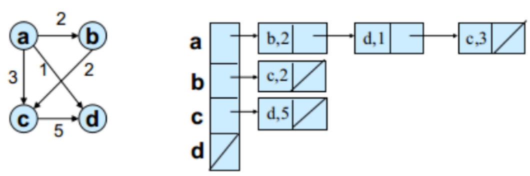
Adjacency Matrix

	0	1	2	3	4	5
0		0.83		0.65		0.51
1	0.83		0.67	0.72	0.46	0.90
2		0.67		0.39		
3	0.65	0.72	0.39		0.24	
4		0.46		0.24		
5	0.51	0.90				



- Symmetric for undirected graphs
- $M[a, a] = 0$, $M[a, b] = \infty$ if a, b are not connected
- Suitable for *Condense Graphs*

Adjacency List



- For undirected graphs, consider every edge to be *doubly directed*
- Suitable for *Sparse Graphs*

Graphs: Traversal (Searching)

Breadth-First Search

Uses a **Queue**, $O(|V| + |E|)$.

1. **Choose a vertex**, mark as `VISITED`, **push into an empty queue**
2. Repeat:
   1. **Pop queue head** v
   2. **For each neighbour of** v**:** u **that is** `NOT VISITED`**:**

        1. Mark u as `VISITED`
        2. Set u's parent to be v
        3. **Push u into queue**
3. **Until queue is empty**
    - If all vertices visited now, then graph is Connected
    - Parent pointers form *a BFS Tree*

Depth-First Search

Uses a **Stack**, $O(|V| + |E|)$.

1. **Choose a vertex**, mark as `VISITED`, **push into an empty stack**
2. Repeat:
    1. If vertex at **stack top v has an `NOT VISITED` neighbour u:**
        1. Mark u as `VISITED`
        2. Set u's parent to be v
        3. **Push u into stack**
    2. **Otherwise, pop v**
3. **Until stack is empty**
    - If all vertices visited now, then graph is Connected
    - Parent pointers form *a DFS Tree*

> Applications of BFS & DFS:
>
> - Finding *Connected Components*
> - Determine **Distances** from source
>   - Use BFS, $s.d = 0$
>   - At every parent setting, $u.d = u.parent.d + 1$
> - Test **Bipartiteness**
>   - Use BFS, alternately set every *layer*
> - Find **Strongly-Connected Components** - *Kosaraju's SCC* Algorithm:
>   1. DFS(G), record *Discovery-time* and *Finish-time*
>   2. **Reverse all edges in G, get G^T**
>   3. DFS(G^T), **pick nodes in decreasing order of Finish-time**
>   4. Each *DFS Tree* formed in second DFS is an SCC!

Topological Sort

An ordering of vertices **in DAG** s.t. v_i is before v_j if there's an edge (v_i, v_j).

- Examples
    - Taking courses in SIST
    - Wearing clothes
- Having Topological Sort ⇔ DAG

In-degree Based Procedure

1. Use an array to store in-degrees of all vertices
2. **Find a vertex v with in-degree 0**
3. Repeat:
    1. Remove v, **update vertices in-degrees**
    2. $v \leftarrow$ a vertex with in-degree 0
        - If none found, not a DAG
4. **Until all vertices picked**

Performance

- Space: $\Theta(|V|)$
- Time:
    - Search through array for in-degree 0: $O(|V|^2)$
    - Every time a vertex's in-degree updated 0, push into queue

- $\Theta(|V| + |E|)$
- Queue initially contains all in-degree 0 vertices
- The array used for queue stores exactly the ordering!

Finding Cirtical Paths

Every node (task) has a computing time.

- **Critical Time**: Minimum time of completing all tasks in parallel
  - Critical time of a task is earliest time that it can be finished
- **Critical Path**: Sequence that determines the minimum time

Task	In-degree	Task Time	Critical Time	Previous Task
A	0	5.2	0.0	∅
B	1	6.1	0.0	∅
C	3	4.7	0.0	∅
D	3	8.1	0.0	∅
E	2	9.5	0.0	∅
F	0	17.1	0.0	∅

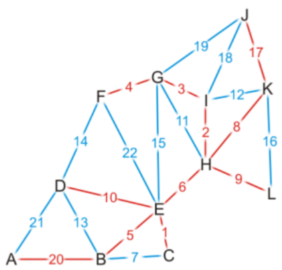
$\rightarrow$

Task	In-degree	Task Time	Critical Time	Previous Task
A	0	5.2	5.2	∅
B	0	6.1	11.3	A
C	0	4.7	31.3	E
D	0	8.1	**39.4**	C
E	0	9.5	26.6	F
F	0	17.1	17.1	∅

1. **Find a vertex v with in-degree 0**
2. Repeat:
   1. Remove v, **update vertices in-degrees**
   2. `v.cirtical_time += v.task_time`
   3. **For every neighbour u of v:**
      - If `v.cirtical_time` $>$ `u.critical_time`:
        1. `u.critical_time` $\leftarrow$ `v.critical_time`
        2. Set `u.previous` $\leftarrow v$
   4. $v \leftarrow$ a vertex with in-degree 0
3. **Until all vertices picked**

Graphs: Minimum Spanning Tree (MST)

- **Spanning Tree**: a subgraph that is a tree, and includes all vertices
  - *Minimum Spanning Tree* is the one with minimum weights
  - Might have Spanning Forest for un-connected graphs
- **Cut Property** (MST Property)



MST is formed by red edges

Prim's Algorithm

Procedure

1. $Vset$ is initially root node $\{s\}$, $Eset$ is initially $\emptyset$
2. R **is initially** $\{(s, u) : \text{for every neighbour } u\}$
3. Repeat:
   1. **Extract the edge** (v, u) **with Minimum weight in** R, where u is the one $\notin Vset$
   2. Add u into $Vset$
   3. Add e into $Eset$
   4. **For every edge** (u, w) **starting from** u
      - **If $w \notin Vset$, add (u, w) into R**
4. **Until** $Vset == V$

Performance

- Space: $O(|V|)$
- Time: $O(|E| \ln |V|)$
  - With Fibonacci Heap: $O(|E| + |V| \ln |V|)$

Kruskal's Algorithm

Procedure

1. $Eset$ is initially $\emptyset$
2. **Make all vertices a disjoint set**
3. **Sort E in non-decreasing order of weights**
4. For every edge (u, v) in E in order:
   - **If u and v are not in same set:**
     1. Add (u, v) into $Eset$
     2. **Union the two sets**

Performance

- Space: $O(|E|)$
- Time: $O(|E| \ln |V|)$

Graphs: Shortest Path

Prerequisites: Weighted Graphs & No *Negative-weighted Loops*.

Dijkstra's Algorithm

Single Source Shortest Path (SSSP), similar to BFS.

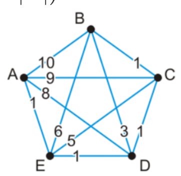
Prerequisites

- **No Negative-weighted Edges**
- Based on Triangle Inequality $\delta(s, u) \leq \delta(s, v) + w(v, u)$

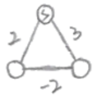
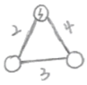
Procedure

1. **Initialize source node** $s.d = 0$
2. **Initialize all other nodes** $v.d = \infty$
3. $Vset$ is initially V
4. Repeat:
   1. **Extract vertex v with minimum d from $Vset$**
   2. Mark v as `VISITED`
   3. **For every neighbour u of v that is `NOT VISITED`:**
      - If $u.d > v.d + w(u, v)$:
        1. $u.d \leftarrow v.d + w(u, v)$
        2. Set u's parent to v
5. **Until $Vset$ is empty**

Performance

- Time: $O(|E| \ln |V|)$
- Worst case - 

Special Cases

- Cannot apply to Negative-weighted Edges: 
- Output is not MST: 

Bellman-Ford Algorithm

Single Source Shortest Path (SSSP), can detect Negative Loops.

Procedure

1. **Initialize source node** $s.d = 0$
2. **Initialize all other nodes** $v.d = \infty$
3. **Repeat** $|V| - 1$ **times:**
    - For every directed edge $(v, u) \in E$:
        - If $u.d > v.d + w(u, v)$:
            1. $u.d \leftarrow v.d + w(u, v)$
            2. Set u's parent to v
4. If there is an edge (v, u) where $u.d > v.d + w(u, v)$:
    - The graph has negative loop

Performance

- Time: $O(|V||E|)$

Floyd-Warshall Algorithm

All Pairs Shortest Paths (APSP).

Procedure

1. **Let** $D^{(0)}$ **be the Adjacency Matrix**
2. Let $Next$ be an Matrix with $Next_{ij} = j$ if there's an edge (i, j)
3. **For** k **from** 1 **to** n**:**
    - **For** i **from** 1 **to** n**:**
        - **For** j **from** 1 **to** n**:**
            - If $D_{ij}^{(k-1)} > D_{ik}^{(k-1)} + D_{kj}^{(k-1)}$:
                1. $Next_{ij} = Next_{ik}$
                2. $D_{ij}^{k} = D_{ik}^{(k-1)} + D_{kj}^{(k-1)}$
            - Else $D_{ij}^{k} = D_{ij}^{(k-1)}$
4. $D^{(|V|)}$ **is the all pairs shortest paths matrix**

> Go through $Next$ Matrix to acquire the shortest path.

Performance

- Time: $O(|V|^3)$

> Modify for finding *Connectiveness*: $T_{ij}^{(k)} = T_{ij}^{(k-1)} \mid\mid (T_{ik}^{(k-1)} \;\&\&\; T_{kj}^{(k-1)})$.

A^* Search

Source-to-Destination Shortest Path.

Prerequisites

Heuristic Function H:

- $G(x) =$ The minimum cost of moving from known paths to x, i.e. d in Dijkstra
- $H(x) =$ Heuristic Function: Estimated cost of moving from x to destination
    - **Admissible Heuristics**: Ensuring $H(x) \leq \delta(x, dest)$

- Only when using Admissible Heuristics can ensure finding optimal Shortest Path
- $F(x) = G(x) + H(x)$

Procedure

1. **Initialize source node to have G-score of 0**
2. **Initialize all other nodes to have G-score of ∞**
3. $Vset$ is initially V
4. Repeat:
    1. **Extract node v with smallest F-score in $Vset$**
    2. **If v is the destination:**
        - **Path found**, END the procedure
    3. Mark v as `VISITED`
    4. **For every neighbour u of v that is** `NOT VISITED`
        - If $G(u) > G(v) + d(v, u)$:
            1. $G(u) \leftarrow G(v) + d(v, u)$
            2. Set $u.parent$ to be v
5. **Until $Vset$ is empty**
6. Assert no path exists

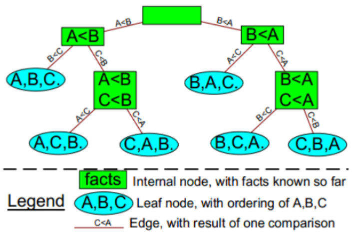
Performance

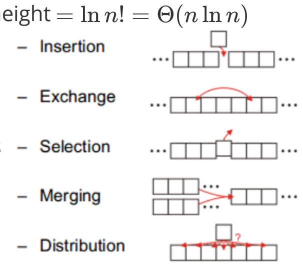
- Time: Depends a lot on $H(x)$
- Degrades to Dijkstra's Alg when $H(x) = C$ for all nodes other than destination (*Discrete Distance*)

Sorting

Taking a list of objects which could be stored in a linear order, returning a reordering s.t. they are in order.

Notations

- In-place / not
    - **In-place**: with the allocation of $O(1)$ memory, ✓
    - **Not In-place**: requires at least $\Theta(n)$ memory
- Run-time
    - **Worst-case** run time: Based on **comparisons**, CANNOT be faster than $\Theta(n \ln n)$
    - **Average-case** run time: Expected
    - **Lower-bound (Best-case)** run time: Must examine each entry at least once, so $\Omega(n)$
- *Comparison Tree* 

    - Any *comparison-based* sorting can be represented by a comparison tree
      - For any array instance, the Sorting procedure is passing through a path from root to a certain leaf
        - # of leaves $= n!$
        - Therefore height $= \ln n! = \Theta(n \ln n)$
- 5 Sorting Techniques 

- **Inversions**: # if pairs that is not in order, $j < k$ but $a_j > a_k$
    - # of inversions $=$
        - Expectedly, half of all pairs are inversions: $\frac{1}{2}\frac{n(n-1)}{2} = \frac{n(n-1)}{4} = O(n^2)$

- Denoted as d

Stupid Sorting Algorithms

- **Bogo Sort**: Randomly order the objects, check if sorted. If not, repeat.
  - In average $\Theta(n \cdot n!)$
- **Bozo Sort**: Check if sorted. If not, randomly swap two entries, and repeat.
  - In average $\Theta(n!)$

Insertion Sort

Given a sorted list of length $k-1$, insert the k-th element into it.

Procedure

```
void insertionSort(Type *array, int const n) {
    for (int k = 1; k < n; ++k) {
        for (int j = k; j > 0; --j) {
            if (array[j - 1] > array[j])
                swap(array[j - 1], array[j]);
            else
                break;
        }
    }
}
```

> With every `swap`, removes an inversion.

Performance

- Space: *In-place*
- Time
  - Worst-case: $O(n^2)$
  - Average-case: $\Theta(n+d)$

Bubble Sort

"Bubble up" the remaining smallest entry every time.

Procedure

```
void bubbleSort(Type *array, int const n) {
    for (int i = 0; i < n - 1; ++i) {
        for (int j = n - 1; j > i; --j ) {
            if (array[j] < array[j - 1])
                swap(array[j], array[j - 1]);
        }
    }
}
```

Performance

- Space: *In-place*
- Time: In all cases $\Theta(n^2)$

Possible improvements

- Store the remaining smallest, avoid so many swaps
- Use `bool` flag to check if no swaps occurred, then already sorted
- Alternate between "Bubbling" and "Sinking"

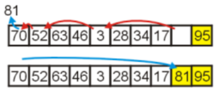
Heap Sort

Build a max heap on array $\to$ Repeatedly extract the root and put it at back.

Procedure

```
void heapSort(Type *array, int const n) {
    max_heap = buildMaxHeapFloyd(array);
    for (int i = n - 1; i > 0; --i)
        array[i] = extractRoot(max_heap);
}
```

Performance

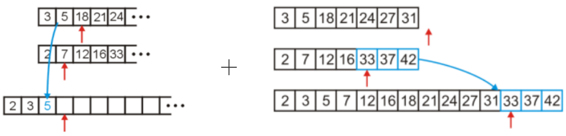
- Space
  - Originally: $\Theta(n)$ for the heap
  - In-place implementation: $\Theta(1)$ 

- Time
  - Worst-case / Averagely: $O(n \ln n)$
  - Best (Most entries are same): $\Theta(n)$

Merge Sort

Divide-and-Conquer: Recursively Divide $\rightarrow$ Merge.

Procedure

- `Merge` Operation: 

  - Analysis
    - Space: $\Theta(n)$
    - Time: $\Theta(n_1 + n_2) = \Theta(n)$

```
void mergeSort(Type *array, int first, int last) {
    if (last - first <= N) {
        insertionSort(array, first, last);
    } else {
        int midpoint = (first + last) / 2;
        mergeSort(array, first, midpoint);
        mergeSort(array, midpoint, last);
        merge(array, first, midpoint, last);
    }
}
```

> Use `insertionSort` for small subarrays (`size <= N`).

Performance

- Space: $\Theta(n)$ for extra array
- Time: In all cases $\Theta(n \ln n)$
  - **Recursion Tree**

Quick Sort

Divide-and-Conquer: Recursively Partition.

Procedure

- `Partition` Operation:

38	10	26	12	43	3	44	80	95	84	66	79	87	96	81

  - Analysis
    - Space: *In-place*
    - Time: $\Theta(n)$

```
void quicksort(Type *array, int first, int last) {
    if (last - first <= N)
        insertion_sort( array, first, last );
```

```
    else {
        Type pivot = array[last]
        int i = first - 1;
        for (int j = first; j < last; ++j) {
            if (array[j] < pivot) {
                ++i;
                swap(array[i], array[j]);
            }
        }
        swap(array[i+1], array[last]);
        quickSort(array, first, i);
        quickSort(array, i+2, last);
    }
}
```

> Use `insertionSort` for small subarrays (`size <= N`).

Performance

- Space: *In-place*, BUT
  - Worst-case: $\Theta(n)$ for function call stacks
  - Average-case: $\Theta(\ln n)$ for function call stacks
- Time
  - Worst-case (`pivot` is always extreme): $O(n^2)$
  - Average-case (`pivot` is well chosen every time): $\Theta(n \ln n)$
    - Use *Median-of-three*: Choose `pivot` as median of first, middle and last element

Possible improvements

Modify the `Partition` procedure to be:

1. **Choose `pivot` using Median-of-three**
   - If `first` is pivot, swap with `middle`
   - If `last` is pivot, swap with `middle`
2. Use two pointers, `low` at 1, `high` at $n-2$
3. Repeat:
   1. **Increment `low` Until `array[low] > pivot`**
   2. **Decrement `high` Until `array[high] < pivot`**
   3. **Swap `array[low]` with `array[high]`**
4. Until `low > high`
5. Put `pivot` at `low`, Put `array[low]` at `n-1`

Bucket Sort

Prerequisites

- Numbers must be in certain range $O(m)$!!
- **Not based on comparisons.**

Procedure

Throw numbers into m buckets $\rightarrow$ Sort inside the buckets $\rightarrow$ Sequentially get numbers from buckets.

> **Counting Sort**: count how many times an "1" occurs...

Performance

- Space: $\Theta(m)$
- Time: $\Theta(n+m)$

Radix Sort

Prerequisites

- Numbers must be digit numbers on certain **base** b (Not necessarily 10)
- Numbers must be finitely long, i.e. in certain range $O(m)$!!

- **Not based on comparisons.**

Procedure

For certain digit numbers, apply *Bucket Sort* on the last digit, then..., finally the first digit.

> Use *Queues* for a Bucket.

Performance

- Space: $\Theta(n)$ (# of buckets $= b$)
- Time: In all cases $\Theta(n \log_b m)$