linear　　nonlinear

linear list　tree　graph　　logical structure

ordered list　linked list　index　hash　　memory structure

- search ; insert ; delete ; merge ; split ; traverse
- string search
- sort : exchange : bubble, quick
  insertion : ..., shell
  selection :
  merge :
  heap : ..., tree selection, tournament.

1. Introduction

1.1 data abstraction & binary relation

a. data abstraction

element + relation → directed, weakly connected graph

b. Cartesian product

$A \times B = \{(a,b) \mid a \in A \ \& \ b \in B\}$

$A_1 \times \cdots \times A_n = \{(a_1, \cdots, a_n) \mid a_i \in A_i\}$

c binary relation

R is a subset of $A \times B$, called an endorelation over A when $A = B$.

{ reflexivity : $(a,a) \in R$, symmetry : $(a,b) \in R \Rightarrow (b,a) \in R$

{ transitivity : $(a,b) \in R, (b,c) \in R \Rightarrow (a,c) \in R$.

1.2 basics of data structure

a. logical structure of data

$B=(D,R)$ D: data, R: endorelation

b. memory structure of data

see page 1

c. abstract data type

data structure + operations

1.3 algorithms

a. brute force:

b. divide and conquer: merge sort, quick sort, shell sort.

c. decrease and conquer: binary search, b-tree search.

d. backtracking: maze

e. greedy: KNP, Prim, kruskal, Dijkstra, Huffman

f. dynamic programming: Floyd-Warshall

2. linear list

$B=(D,R)$ $D=\{a_i \mid i=1,\dots,n\}$, $R=\{(a_i,a_{i+1}) \mid i=1,\dots,n-1\}$

2.1 ordered list

2.2 stack

double stack:

e.g. recursion + divide & conquer

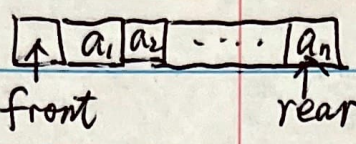
$T(n)=aT(\frac{n}{b})+f(n)$

$\begin{cases} c<\log_b a \\ c\sim\log_b a \\ c>\log_b a \end{cases}$

```
Divide(P){ // solve problem P of size n
    if (P solvable) solve P directly
    else for(i=1:k) y_i = Divide(P_i) //solve subproblem
    return merge(y_1,... y_k) // merge solutions to subproblems
```

2.3 queue

a. sequential



b. circular delete: front = rear = 0

$$\begin{cases} \text{enqueue: } (rear+1)\%ms \\ \text{dequeue: } (front+1)\%ms \end{cases}$$

$$\begin{cases} \text{empty: } front == rear \\ \text{full: } (rear+1)\%ms == front \end{cases} \Big\} \text{waste one unit}$$

c. priority: heap

2.4 array and matrix

a. row-based array: $AD(a_{ij}) = AD(a_{11}) + [(i-1)n + j - 1] \cdot unit$;

Column-based array: $AD(a_{ij}) = AD(a_{11}) + [(j-1)n + i - 1] \cdot unit$

b. compressed matrix

- lower-triangular:

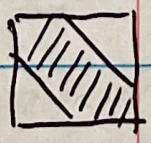
row: $k = \frac{1}{2}i(i-1) + j$

col: $k = \frac{[2n-(j-1)]j}{2} + i - (j-1)$

- symmetric → lower-triangular

$$k = \begin{cases} \frac{1}{2}i(i-1) + j & i >= j \\ \frac{1}{2}j(j-1) + i & i < j \end{cases}$$

- diagonal:



$$k = \begin{cases} (\lfloor \frac{m}{2} \rfloor + 1) \cdot i + j - \lfloor \frac{m}{2} \rfloor + 1 & \text{else} \\ 0 & \text{if not } (i \le j \,\&\, j \le i + \lfloor \frac{m}{2} \rfloor) \,||\, (i > j \,\&\, i \ge j + \frac{m}{2}) \end{cases}$$

c. sparse matrix: B(row, col, val)

$$\begin{cases} POS[k]: \text{first non-0 element in row } k & POS[1] = 0 \\ NUM[k]: \text{num of non-0 elements in row } k & NUM[B[i,1]] = NUM[B[i-1]] + 1 \end{cases}$$

3. linked list

3.1 regular linked list

insertion: empty list; insert to the 1st node; can't find where to insert

deletion: empty list; delete the 1st node; can't find where to delete

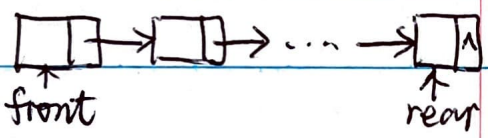
merge: choose a base list, $O(m+n)$

split: odd pos / even pos : $O(n)$

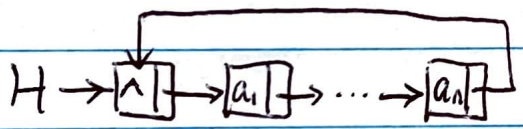
3.2 linked stack, linked queue

a. linked stack

b. linked queue



3.3 circular linked list



empty: 

: no need to single out the empty case (v.s 3.1)

check rear: $p \to next == head$

merge: $p_1 \to next = head_2$, $p_2 \to next = head_1$

split: create new head, consider odd pos / even pos, choose base

3.4 multi-linked list

3.5 lists.

linear; recursive definition; depth = # of layers of recursion

length = # of "next".

e.g.

head:	flag = 0	sublist	^
atom:	flag = 1	data	next
sublist:	flag = 2	sublist	next

4. tree : binary tree

4.1 basics of tree

n: number of nodes, degree of n_i: d_i $\Rightarrow$ $n = \sum_{i=1}^{n} d_i + 1$

k: branches of a node (maximum) : least depth $= \lceil \log_k [n(k-1)+1] \rceil$

4.2 binary tree

a. basic properties

- n_0 leaves, n_2 nodes with degree 2 $\Rightarrow$ $n_0 = n_2 + 1$

  proof: let n be # of nodes, n_1 be # of nodes with degree 1

  $$n = n_0 + n_1 + n_2 \quad ①$$

  every node has a parent except the root:

  $$n - 1 = 2n_2 + n_1 \quad ②$$

  ① ② $\Rightarrow$ $n_0 = n_2 + 1$

- full (all leaves present) vs balanced vs complete (leaves present l→r)

  n nodes $\Rightarrow$ depth $= \lfloor \log_2 n \rfloor + 1$

  depth h $\Rightarrow$ $n \leq 2^h - 1$

- left child of i : $2i+1$, right child of i : $2i+2$

  parent : $\lfloor (i-1)/2 \rfloor$ $\quad i = 0, 1, \cdots$ (for heap; space efficient)

  left child of i : $2i$, right child of i : $2i+1$

  parent : $\lfloor i/2 \rfloor$ $\quad i = 1, 2, \cdots$ by mathematical induction

4.3 memory structure of binary tree

linked list : complete binary trees can be stored using a heap

4.4

- pre-order : visit → l → r
- in-order : l → visit → r
- post-order : l → r → visit

} DFS, time : $O(n)$, space : $O(\log_2 n) \sim O(n)$

can also be implemented in a non-recursive way

→ search : get lists representation

→ for deletion, getting depth, # of nodes

4.5 BFS binary tree time: $O(n)$ space: k^{h-1}, k: degree, h: depth

level order tree traversal: ~~for getting depth, deletion, getting # of nodes~~

while(!EmptyQueue()){enqueue left child; enqueue right child; dequeue;}

4.6 threaded binary tree for $O(1)$ space ~~in order~~ traversal:

l_tag : l_kid : data : r_kid : r_tag

- threaded preorder: thread cur_node & preorder predecessor → l→r
- threaded inorder: l→thread cur_nod & inorder predecessor → r
- threaded post order: l→r→thread cur_node & postorder predecessor

predecessor → l_kid → r_kid.

4.7 counting in binary trees

number of inorder traversals corresponding to a preorder: $\frac{1}{n+1}C_{2n}^{n}$

to specify a binary tree: using preorder to determine root, inorder kids.

4.8 applications of binary trees

a. ordered tree → binary linked list → binary tree

b. optimal binary tree (huffman tree)

depth h: $2h-1 \le n \le 2^h-1$

$$\begin{cases} n_0 = n_2+1 \\ n_0+n_2 = n \end{cases} \Rightarrow \begin{cases} n_0 = \frac{n+1}{2} \\ n_2 = \frac{n-1}{2} \end{cases}$$

minimize: $\sum_{i=1}^{n} \text{weight}(i) \cdot \text{length}(i)$

algorithm: select the two trees from the forest with minimum weights to from a binary tree; greedy

root weight = l_weight + r_weight

application: huffman code (non-prefix code): left 0 right 1

c. binary search tree BST

depth h: $h < n < 2^h - 1$

- n_left < n_right : in order insertion, time, space: $O(\log_2 n) \sim O(n)$
- deletion { leaf: delete directly
  degree 1 node: delete following child
  degree 2 node: replace with in order predecessor → delete in order predecessor { leaf ....
  degree 1 ....

d. heap (binary sort) findmin/findmax in O(1)

depth h: $2^{h-1} \le n \le 2^h - 1$

{ max-heap: parent > kid
  min-heap: parent < kid } list, complete binary tree index by layer

- insertion: insert to rear (lower right), sift up the element: $O(\log_2 n)$
- deletion: { directly delete if at rear
  swap top & rear, sift down the top: $O(\log_2 n)$ if at top
- creation (sorting): { insert each element (sift up): $O(n \log_2 n)$ ⊕
  sift down each element from lower right to upper left: $O(n)$

e. solution space tree

backtracking = pre order traversal of the solution space tree

4.9. Equivalence class and union-find (disjoint-set)

memory structure: multi-branch linked list-based tree

{ make set: rank = 0, parent = x
  find: x.parent = Find(x.parent) : recursive ←
  union: make the higher-ranked parent as parent. ←

application: partition a set to get equivalence classes; Kruskal — MST

5. graph

5.1 representation

$G = (V, E)$ $\quad G' \leq G$ if G' is a subgraph of G

undirected: degree = edges connected

directed: degree = inbound edges + outbound edges

$\sum_{v \in V} \deg(v) = 2|E|$

$n-1 \leq |E| \leq C_n^2$ for undirected graphs

$n \leq |E| \leq 2C_n^2$ for directed graphs

5.2 memory

a. adjacency matrix for dense graphs

inbound degree $= \sum_{i=0}^{n-1} col_i = \sum_{i=0}^{n-1} A[i][j]$

outbound degree $= \sum_{j=0}^{n-1} row_j = \sum_{j=0}^{n-1} A[i][j]$

b. linked list for sparse graphs

{ triplet representation : (row, column, value)

linked representation : (row, column, value, next node)

5.3 traversal

a. DFS (DFT): adjacency matrix : $O(n^2)$, linked list : $O(n+e)$, space : $O(n)$

≈ pre order traversal of a tree

b. BFS (BFT): adjacency matrix : $O(n^2)$, linked list : $O(n+e)$, space : $O(n)$

≈ level order traversal of a tree (single source single sink shortest path in an unweighted graph)

5.4 MST minimum spanning tree ($\min \sum_{i=0}^{n-1} e_i$, $e_i \in MST$)

$V(G') = V(G)$, n vertices, $n-1$ edges.

a. Prim : $O(n^2)$: greedy, use heap

b. Kruskal : $O(e \log e)$: greedy, use union-find structure

5.5 Single source shortest path (all sinks)

a. w/o negative weights : Dijkstra, based on priority queue
$O((|V|+|E|)\log|V|)$ faster than iterative DFS's

b. w/ negative weights : Bellman-Ford, slower than Dijkstra
$O(|V|\cdot|E|)$

c. SPFA : shortest path faster algorithm, w/ negative weights
$O(k\cdot|E|)$, $k \ll |V|$

5.6 acyclic shortest path, DAGs
$O(|V|+|E|)$

5.7 topological sorting, DAGs
$O(|V|+|E|)$, using DFS.

5.8 all-source shortest path (all sinks), w/ negative weights.
$O(|V|^3)$, Floyd-Warshall, dynamic programming

6. Search, string search

6.1 binary search tree : see 4.8, C, worst case $O(n)$ for search, insert, delete ~~and space~~.

6.2 self-balancing binary search tree

a. AVL tree } worst case $O(\log n)$ for
b. red-black tree } search, insert, delete.

6.3 high fan-out self-balancing tree

B-tree, $O(\log n)$ for large blocks of data search, insert ...

6.4 string searching algorithm. string : n, pattern : m

a. straightforward : $O(m\cdot n)$ → Compute matching prefix & suffix

b. Knuth-Morris-Pratt : preprocessing : $O(m)$, matching : $O(n)$, space : $O(m)$

7. Sort

7.1 exchange sort : swap.

a. bubble sort → bidirectional bubble sort $O(n^2)$

b. quick sort : choose the median of $R[m]$, $R[\frac{m+n}{2}]$, $R[n]$, divide and conquer, recursion $O(n\log n) \to O(n^2)$

7.2 insertion sort : insert to sorted list

a. naive $O(n^2)$

b. binary-search the sorted list $O(n\log n) \to O(n^2)$ arr swaps

c. shell $O(n^{1.5})$

7.3 Selection sort = select from unsorted list

a. naive $O(n^2)$

b. heap sort: time $O(n) - O(n\log n)$, space: $O(1)$

see 4.8 d, divide and conquer.

7.4 merge sort : divide and conquer, $O(n\log n)$

{ top-down $O(n)$ space. external sort ("disk")

bottom-up easy to parallelize, can be used on

7.5 distribution sort (non-comparison sort) integers

a. bucket sort time: $O(n+k) - O(n^2)$, space: $O(n+k)$

b. counting sort time: $O(n+k)$, space: $O(n+k)$

c. radix sort. time: $O(wn)$, space: $O(w+N)$