FDS

jyt555/zju_cs: My note warehouse (github.com): `https://github.com/jyt555/zju_cs`

1 Scores

- **lecture grade(75) = homework exercises(10) + quizzes(10) + mid-term exam(15) + final exam(40)**
  [the score of mid-term exam can be covered by that of final exam]
- laboratory grade(25 / 30)
- homework PTA
- project : independently
  - submit initial version for peer review(1week)
  - participate in peer review(2days)
  - revise paper and submit to TA(2days)
  - receive final grading from TA

---

2 algorithm analysis

2.1 definition

input: 0 or more
output: at least one
definiteness: clear and unambiguous
finiteness: the algorithm terminates after finite number of steps
effectiveness: be basic enough to be carried out and feasible(可行的)

A **program** is written in some programming language, and does not have to be finite
An **algorithm** can be described by human languages, flow charts, some programming languages, or pseudocode.

2.2 what to analyze

- machine & compiler-dependent **run times**
- **time & space complexities**: machine & compiler-independent

typically the following two functions are analyzed:
$\mathbf{T_{avg}(N)}$ **&** $\mathbf{T_{worst}(N)}$ — the average and worst case time complexities as functions of input size N (if there is more than one input, these functions may have more than one argument)

```c
void add(int a[][MAX_SIZE],
         int b[][MAX_SIZE],
         int c[][MAX_SIZE],
         int d[][MAX_SIZE],
         int rows, int cols)
{
    int i, j;
    for (i=0;i<rows;i++)        // rows + 1
        for (j=0;j<cols;j++)    // rows(cols+1)
            c[i][j] = a[i][j] + b[i][j];    // rows*cols
}
// T(rows, cols) = 2*rows*cols + 2*cows + 1
```

2.3 asymptotic(渐近线的) notation(O,Ω,Θ,o)

the point of counting the step is to predict the growth in run times as the N change, and thereby compare the time complexities of two programs.

2.3.1 definition:

T(N) = O(f(N)): T(N) <= c·f(N) and c > 0 [always take the smallest f(N)]
T(N) = Ω(g(N)): T(N) >= c·g(N) and c > 0 [always take the largest g(N)]
T(N) = Θ(h(N)): T(N) = O(h(N)) and T(N) = Ω(h(N))
T(N) = o(p(N)): T(N) = O(p(N) but T(N) ≠ Ω(p(N))

Little-oh is different from Big-Oh, because Big-Oh allows the possibility that the growth rates are the same.(omega theta)

2.3.2 rules of asymptotic notation:

if $T_1(N) = O(f(N))$ and $T_2(N) = O(g(N))$, then

$\mathbf{T_1(N) + T_2(N) = max\{ O(f(N)), O(g(N)) \}}$

$\mathbf{T_1(N) * T_2(N) = O(f(N)*g(N))}$

if T(N) is a polynomial(多项式的) of degree k, then $\mathbf{T(N) = \Theta(N^k)}$

$\mathbf{\log^k N = O(N)}$ –> logarithms grows very slowly

```c
void add(int a[][MAX_SIZE]
         int b[][MAX_SIZE]
         int c[][MAX_SIZE]
         int d[][MAX_SIZE]
         int rows, int cols)
{
    int i, j;
    for(i=0;i<rows;i++)         // 0(row)
        for(j=0;j<cols;j++)     // 0(rows·cols)
            c[i][j] = a[i][j] + b[i][j];    // 0(rows·cols)
}
// T(rows, cols) = 0(rows, cols)
```

2.3.3 [general rules]:

for loops:
the running time of the statements inside the for loop, times the number of iterations

nested(嵌套的) for loops:
the running time of the statements multiplied by the product of the size of all the for loops

consecutive satements:
add (the maximum is the one that counts)

if / else:
    if(condition) S1;
    else S2;
the running time is never more than the running time of the test plus the larger of the running time of S1 and S2.

```
// Fibonacci number:
long int Fib(int N)         // T(N)
{
    if(N<=1)               // O(1)
        return 1;          // O(1)
    else
        return Fib(N-1) + Fib(N-2);
        // O(1)   T(N-1)   T(N-2)
}
// T(N) gorwss exponentially, but the space complexity is O(N)
```

2.4 compare the algorithms

lg: Given (possibly negative) integers $A_1, A_2, \ldots A_n$, find the maximum value of $\sum^{j}_{k=i} A_k$

2.4.1 divide and conquer 分而治之

T(N) = 2 * T(N/2) + c * N ==> O(NlogN)

2.4.2 On-line Algorithm: O(N)

```
int MaxSubsequenceSum(const int A[], int N)
{
    int ThisSum, MaxSum, j;
    ThisSum = MaxSum = 0;
    for(j = 0;j < N;j++)
    {
        ThisSum += A[j];
        if(ThisSum > MaxSum)
            MaxSum = ThisSum;
        else if(ThisSum < 0)
            ThisSum = 0;
    }
    return MaxSum;
}
```

At any point in time, the algorithm can correctly give an answer to the subsequence problem for the data it has already read (the other algorithms do not share this property). Algorithms that can do this are called on-line algorithms. An **on-line algorithm** that requires only constant space and runs in linear time is just about as good as possible.

2.5 logarithms in the Running Time

binary search: O(logN)

```
int BinarySearch(const ElementType A[],
                        ElementType X, int N)
{
    int Low, Mid, High;
    Low = 0; High = N - 1;
    while(Low <= High)
    {
        Mid = (Low + High) / 2;
        if(A[Mid] < X)
            Low = Mid +  1;
        else if(A[Mid] > X)
                High = Mid - 1;
        else return Mid;    // found
    }
    return NotFound;    //defined as -1
}
```

2.6 checking your analysis

when T(N) = O(N), check if T(2N)/T(N) ≈ 2 …

---

3 lists, stacks, and queues

3.1 abstruct data type (ADT)

Data Type = {Objects} ∪ {Operations}

lg: int = { 0, ±1, ±2, … INT_MAX, INT_MIN } ∪ { +, -, *, /, %, …}

An **Abstract Data Type (ADT)** is a data type that is organized in such a way that the specification on the objects and specification of the operations on the objects are separated from the representation of the objects and the implementation on the operations.

3.2 the list ADT

3.2.1 array[i] = $item_i$:

:) Find_Kth takes O(1) time

:(MaxSize has to be estimated

:(Insertion and Deletion not only take O(N) time, but also involve a lot of data movements which takes time

3.2.2 linked lists :

:(Find_Kth takes O(N) time

:) Insertion and Deletion take O(1) time

(add `a dummy head node` to a list, to delete the first node from a list)

3.3 the stack ADT

Last-In-First-OUT (**LIFO**, reverse the order)

needs one pointer

linked list implementation and array implementation

infix / prefix / postfit expression

- a - b - c –> a b - c -, but **2 ^ 2 ^ 3 (2 ^ (2^3)) –> 2 2 3 ^ ^**

3.4 the queue ADT

FIFO

needs two pointers

array implementation

circlular queue

---

4 tree

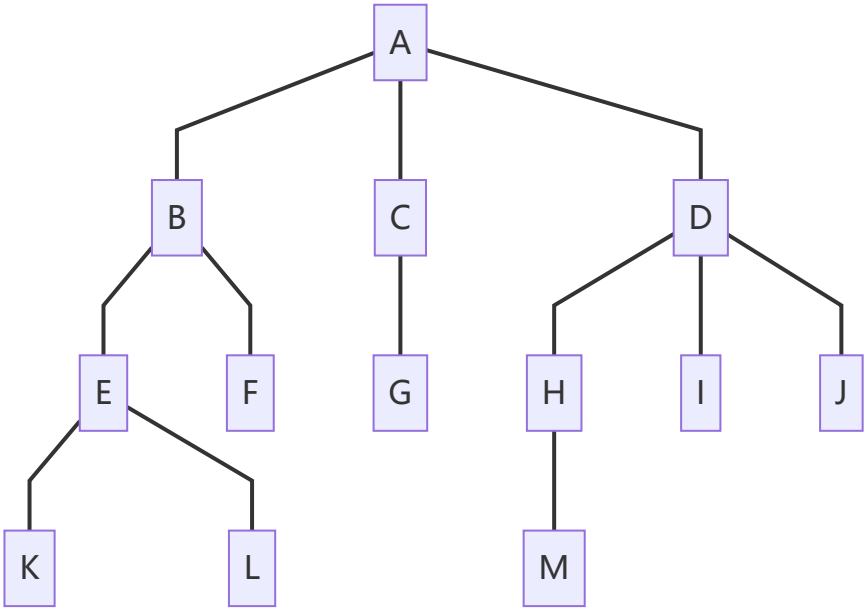
4.1 preliminaries（预备知识）

Terminology（专有名词）

A **tree** is a collection of nodes. The collection can be empty; otherwise, a tree consists of
(1) a distinguished node **r**, called the **root**
(2) and zero or more nonempty **(sub)tree**, each of whose roots are connected by a directed **edge** from r.

- degree of a node: number of subtrees of the node
- degree of a tree: the maximum of degree of nodes
- parent: a node that has subtrees
- children: the roots of the subtrees of a parent
- siblings: children of the same parent
- leaf (terminal node): a node with degree 0
- path from n_1 to n_k
- length of path
- **depth** of n_1: length of the unique path **from the root to n_i**
- **height** of n_1: length of the longest path **from n_i to a leaf,** height(leaf) = 0
- ancestors of a node
- descendants of a node

Implementation:



List Representation: (A(B(E(K,L),F),C(G),D(H(M),I,J)))

FirstChild-NextSibling Representation: Element | FirstChild & NextSibling

4.2 Binary Trees

Element | Left & Right

Expression Trees(syntax trees 语法树)

4.2.1 Tree Traversals —— visit each node exactly once

know **preorder and postorder can't determine the inorder**
(but preorder & inorder -> postorder; postorder & inorder -> preorder)

```
/* Preorder */
void preorder(tree_ptr tree)
{   if(tree){
        visit(tree);
        for(each child C of tree)
            preorder(C);
    }
}

/* Postorder */
void postorder(tree_ptr tree)
{ if(tree){
    for(each child C of tree)
        postorder(C);
    visit(tree);
    }
}

/* Levelorder */
void levelorder(tree_ptr tree)
{ enqueue(tree);
```

```
  while(queue is not empty){
      visit(T = dequeue());
      for(each child C of T)
          enqueue(C);
  }
}

/* Inorder */
/* only for binary tree */
void inorder(tree_ptr tree)
{ if(tree){
    inorder(tree->Left);
    visit(tree->Element);
    inorder(tree->Right);
  }
}
/* Iterative Program */
void iter_inorder(tree_ptr tree)
{
    Stack S = CreateStack(MAX_SIZE);
    for( ; ; ){
        for( ; tree; tree = tree->Left)
            Push(tree, S);
        tree = Top(S); Pop(S);
        if(!tree) break;
        visit(tree->Element);
        tree = tree0>Right;
    }
}
```

```
static void ListDir(DirOrFile D, int Depth)
{
    if(D is a legitimate entry){
        printName(D, Depth);
        if(D is a directory)
            for(each child C of D)
                ListDir(C, Depth+1);
    }
}
/* Note: Depth is an internal variable and must not be
 * seen by the user of this routine.
 * One solution is to define another interface
 * function as the following:
 */
void ListDirectory (DirOrFile D)
{   ListDir(D, 0);                }
```

Threaded Binary Tree:

A binary tree with n nodes has (n+1) empty pointer

- how to use it: Threaded Binary Tree

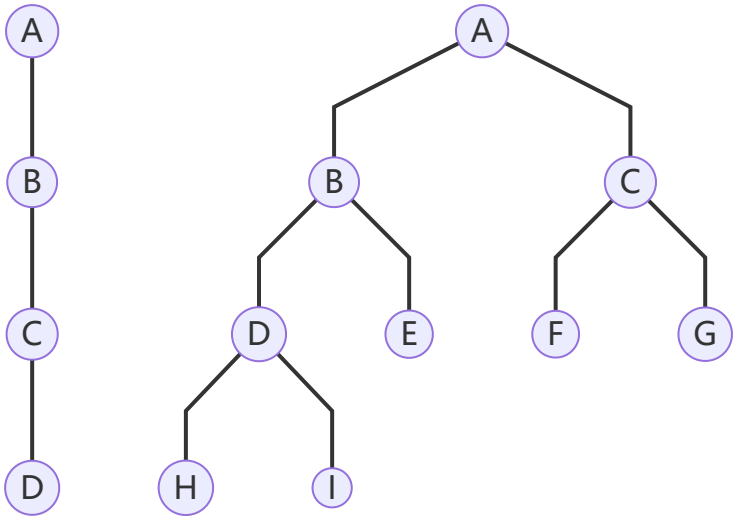
–

in a binary tree, left child and right child are different.

skewed (斜的) binary tree

Complete Binary Tree | all the leaf nodes are on two adjacent levels

> 一棵深度为k的有n个结点的二叉树，对树中的结点按从上至下、从左到右的顺序进行编号，如果编号为i（1≤i≤n）的结点与满二叉树（又称完美二叉树）中编号为i的结点在二叉树中的位置相同，则这棵二叉树称为完全二叉树。
>
> 完全二叉树的特点：叶子结点只能出现在最下层和次下层，且最下层的叶子结点集中在树的左部。需要注意的是，满二叉树肯定是完全二叉树，而完全二叉树不一定是满二叉树。



4.2.2 properties of binary trees:

- the maximum number of nodes on level i is 2^{i-1}, i>=1

  the maximum number of nodes in a binary tree of depth k is $2^{k+1}-1$, k>=1

- for any nonempty binary tree, $n_0 = n_2 + 1$ where n_0 is the number of leaf nodes and n_2 is the number of nodes of degree 2

```
n = n0 + n1 + n2, n = B + 1, B = n1 + 2 * n2
```

4.3 Search Tree ADT

- every node has a key (integer), and the keys are distinct.
- The keys in a nonempty left subtree must be smaller than the key in the root of the subtree.
- The keys in a nonempty right subtree must be larger than the key in the root of the subtree.
- The left and right subtrees are also binary search trees.

ADT:
Objects: A finite ordered list with zero or more elements.
Operations:
 SearchTree MakeEmpty(SearchTree T);
 Position Find(ElementType X, SearchTree T);
 Position FindMin(SearchTree T);
 Position FindMax(SearchTree T);

- SearchTree Insert(ElementType X, SearchTree T);
- SearchTree Delete(ElementType X, SearchTree T);
- ElementType Retrieve(Position P);

```c
/* find */
Position Find(ElementType X, SearchTree T)
{
    if(T == NULL)
        return NULL; /* not found in an empty tree */
    if(x < T->Element)
        return Find(X, T->Left); /* search left subtree */
    else if(X > T->Element)
        return Find(X, T->Right); /* search right subtree */
    else return T; /* found */
}
/* T(N) = S(N) = O(d),where d is the depth of X */

// line 7 and line 9 are tail recursions, so:
Position Iter_Find(ElementType X, SearchTree T)
{
    while(T){
        if(X == T->Element) return T; /* found */
        if(X < T->Element) T = T->Left;
        else T = T->Right;
    }
    return NULL; /* not found */
}
```

```c
/* FindMin */
Position FindMin(SearchTree T)
{
      if (T == NULL)
          return NULL; /* not found in an empty tree */
      else
          if(T->Left == NULL)   return T;  /* found left most */
          else return FindMin(T->Left);   /* keep moving to left */
}
/* FindMax */
Position FindMax(SearchTree T)
{
      if(T != NULL)
          while(T->Right != NULL)
            T = T->Right;   /* keep moving to find right most */
      return T;  /* return NULL or the right most */
}
/* T(N) = O(d)
```

```c
/* Insert */
SearchTree Insert(ElementType X, SearchTree T)
{
    if (T == NULL){ /* Create and return a one-node tree */
        T = malloc(sizeof(struct TreeNode));
        if (T == NULL)
            FatalError("Out of space!!!");
        else {
            T->Element = X;
```

```
            T->Left = T->Right = NULL;
        }
     }  /* End creating a one-node tree */
     else  /* If there is a tree */
        if (X < T->Element)
            T->Left = Insert(X, T->Left);
        else
            if(X > T->Element)
                T->Right = Insert(X, T->Right);
       /* Else X is in the tree already; we'll do nothing */
    return T;   /* Do not forget this line!! */
}
/* T(N) = O(d) */
```

```
/* Delete */
/* delete a leaf node: reset it parent link to NULL */
/* delete a degree 1 node: replace the node by its single child */
/* delete a degree 2 node:
        1: replace the node by the largest one in its left subtree or the smallest one in its right subtree.
        2: delete the replacing node from the subtree. */
SearchTree Delete(ElementType X, SearchTree T)
{
    Position  TmpCell;
    if (T == NULL)   Error("Element not found");
    else if (X < T->Element)  /* Go left */
        T->Left = Delete(X, T->Left);
    else if (X > T->Element)  /* Go right */
        T->Right = Delete(X, T->Right);
    else  /* Found element to be deleted */
        if (T->Left && T->Right) {  /* Two children */
          /* Replace with smallest in right subtree */
            TmpCell = FindMin(T->Right);
            T->Element = TmpCell->Element;
            T->Right = Delete(T->Element, T->Right);} /* End if */
    else{  /* One or zero child */
            TmpCell = T;
            if (T->Left == NULL) /* Also handles 0 child */
                T = T->Right;
            else if (T->Right == NULL)  T = T->Left;
                 free(TmpCell);}  /* End else 1 or 0 child */
    return T;
}
/* T(N) = O(h), where h is the height of the tree */

/* 'free' costs much, so how can we do ? */
/* lazy deletion */
/* Add a flag field to each node, to mark if a node is active or is deleted. Therefore we can delete a node without actually freeing the space of that node. If a deleted key is reinserted, we won't have to call malloc again. */
```

average-case analyse: log n

---

5 Priority Queues (Heaps)

—- delete the element with the highest / lowest priority

5.1 ADT Model

objects: a finite ordered list with zero or more elements

operations:

- PriorityQueue Initialize(int MaxElements);
- void Insert(ElementType X, PriorityQueue H);
- ElementType DeleteMin(PriorityQueue H);
- ElementType FindMin(PriorityQueue H);

5.2 simple implementations

- array:
  - Insertion — add one item at the end ~ θ(1)
    Deletion — find the largest \ smallest key ~ θ (n)
    remove the item and shift array ~ O(n)
- linked list:
  - Insertion — add to the front of the chain ~ θ (1)
    Deletion — find the largest \ smallest key ~ θ (n)
    remove the item ~ θ(1)
- ordered array:
  - Insertion — find the proper position ~ O(n)
    shift array and add the item ~ O(n)
    Deletion — remove the first \ last item ~ θ(1)
- ordered linked list:
  - Insertion — find the proper position ~ O(n)
    add the item ~ θ(1)
    Deletion — remove the first \ last item ~ θ(1)
- binary search tree: not balanced

5.3 binary heap

full binary tree: 除了叶子节点，每个节点都有两个子节点；

complete binary tree: 叶子节点均在最后一层的最左边；

perfect binary tree: 每层都填满，即是完满二叉树也是完全二叉树（也叫满二叉树）

5.3.1 structure property:

A binary tree with n nodes and height h is **complete** iff its nodes correspond to the nodes numbered from 1 to n in the perfect binary tree of height h.

A complete binary tree of height h has between 2^h and 2^{h+1} - 1 nodes.

Array representation: BT[n + 1] (BT[0] is not used)

$$(1)\ index\ of\ parent(i) = \begin{cases} \lfloor i/2 \rfloor, & if\ i \neq 1 \\ None, & if\ i = 1 \end{cases}$$

$$(2)\ index\ of\ left'child(i) = \begin{cases} 2i, & if\ 2i \leq n \\ None, & if\ 2i > n \end{cases}$$

$$(3)\ index\ of\ right'child(i) = \begin{cases} 2i+1, & if\ 2i+1 \leq n \\ None, & if\ 2i+1 > n \end{cases}$$

```
PriorityQueue Initialize(int  MaxElements)
{
     PriorityQueue H;
     if(MaxElements < MinPQSize)
        return Error("Priority queue size is too small");
     H = malloc(sizeof(struct HeapStruct));
     if (H == NULL)
        return FatalError("Out of space!!!");
     /* Allocate the array plus one extra for sentinel (哨兵)*/
     H->Elements = malloc((MaxElements + 1) * sizeof(ElementType));
     if(H->Elements == NULL)
        return FatalError("Out of space!!!");
     H->Capacity = MaxElements;
     H->Size = 0;
     H->Elements[0] = MinData;  /* set the sentinel */
     return H;
}
```

5.3.2 Heap order property

A **min tree** is a tree in which the key value in each node is no larger than the key values in its children (if any). A **min heap** is a complete binary tree that is also a min tree.

Analogously, we can declare a **max heap** by changing the heap order property.

5.3.3 basic heap operations (⭐ important)

```
/* H->Element[0] is a sentinel */
void  Insert( ElementType  X,  PriorityQueue  H )
{
    int i;
    if(IsFull(H)){
        Error("Priority queue is full");
        return;
    }
    for(i=++H->Size; H->Element[1/2] > x; i/=2)
        /* percolate up 逐渐上升 */
        H->Elements[i] = H->Elements[i/2];
        /* faster than swap */
```

```
    H->Elements[i] = x;
}
/* T(N) = O(log N) */
```

```
ElementType DeleteMin(PriorityQueue H)
{
    int i, Child;
    ElementType MinElement, LastElement;
    if(IsEmpty(H)){
         Error("Priority queue is empty");
         return H->Elements[0];}
    MinElement = H->Elements[1];  /* save the min element */
    LastElement = H->Elements[H->Size--];  /* take last and reset size */
    for(i=1; i*2 <= H->Size; i = Child){  /* Find smaller child */
         Child = i * 2;
         if(Child != H->Size && H->Elements[Child+1] < H->Elements[Child])
           Child++;
         if(LastElement > H->Elements[Child])   /* Percolate one level */
           H->Elements[i] = H->Elements[Child];
         else break;   /* find the proper position */
    }
    H->Elements[i] = LastElement;
    return MinElement;
}
/* T(N) = O(log N) */
```

5.3.4 other heap oprations

finding any key except the minimum one will have to take a linear scan through the entire heap.

- DecreaseKey(P, Δ, H)

  Lower the value of the key in the heap H at position P by a positive amount of Δ……so my programs can run with highest priority.

  percolate up

- IncreaseKey(P, Δ, H)

  percolate down

- Delete(P, H)

  DecreaseKey(P, ∞, H); DeleteMin(H)

  Remove the node at position P from the heap H …… delete the process that is terminated (abnormally) by a user.

- BuildHeap(H)

  Perculate down

  T(N) = O(N)

【Theorem】For the perfect binary tree of height h containing 2^{h+1} - 1 nodes, the sum of the heights of the nodes is $\mathbf{2^{h+1} - 1 - (h + 1)}$.

$$\sum_{k=1}^{n} log\ k \quad ?$$

$$\sum_{k=1}^{n} ln\ k = \int_{1}^{n} lnx\ dx = (xlnx - x)|_{1}^{n}$$

5.4 applications of priority queues

5.5 d-Heaps —— all nodes have d children

- DeleteMin will take d - 1 comparisons to find the smallest child. Hence the total time complexity would be O(d $\log_d$ N).
- *2 or /2 is merely a bit shift, but *d or /d is not.
- When the priority queue is too large to fit entirely in main memory, a d-heap will become interesting.

---

6 Disjoint set

6.1 equivalence relations

A relation R is defined on a set S if for every pair of elements (a, b), a, b∈S, a R b is either true or false. If a R b is true, then we say that a is related to b.

A relation, ~, over a set, S, is said to be an **equivalence relation** over S iff it is **symmetric, reflexive, and transitive** over S.

Two members x and y of a set S are said to be in the same **equivalence class iff x ~ y.**

6.2 dynamic equivalence problem

{2,4,7,11,12}, {1,3,5}, {6,8,9,10}

```
Algorithm:
{   /* step 1: read the relations in */
    Initialize N disjoint sets;
    while(read in a ~ b){
        if(!(Find(a) == Find(b)))
    Union the two sets;
    } /* end-while */
    /* step 2: decide if a ~ b */
    while(read in a and b)
        if(Find(a) == Find(b))   output(true);
        else output(false);
}

```

Operations :

(1) Union(i, j) ::= Replace S_i and Sj by S = S_i U S_j

(2) Find(i) ::= Find the set S_k which contains the element i.

6.3 Basic data structure

Union(i, j)

Make Si a subtree of S_j , or vice versa. That is, we can set the parent pointer of one of the roots to the other root.

1’ pointer

2’ S[element] = the element’s parent, S[root] = 0 and set name = root index

etc. S[4] = 10;

[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]	[10]
4	0	2	*10*	2	10	10	10	4	0

```
void SetUnion(DisjSet S, SetType Rt1, SetType Rt2)
{    S [Rt2] = Rt1 ;     }
```

Find(i)

1’ find(i) = ‘S’

2’

```
SetType Find(ElementType X, DisjSet S)
{   for ( ; S[X] > 0; X = S[X]);
    return X;
}
```

a worst case: T = $O(N^2)$

6.4 Smart Union Algorithms (⭐)

6.4.1 Union-by-Size: always change the smaller tree

```
S[Root] = -size; /* initialized to be -1 */
```

【Lemma】Let T be a tree created by union-by-size with N nodes, then

$$herght(T) \leq \lfloor log_2\ N \rfloor + 1$$

proof: induction. (Each element can have its set name changed at most $\log_2$N times)

Time complexity of N Union and M Find operations is **O(N + M·$\log_2$N)**

6.4.2 Union-by-Height: always change the shallow tree

This is an easy algorithm, since the height of a tree increases only when two equally deeptrees are joined (and then the height goes up by one). Thus, union-by-height is a trivial modification of union-by-size.

6.4.3 Union-by-rank

(union by height with path compression)

```c
/* assume root1 and root2 are roots */
/* union is a C keyword, so this routine is named set_union */
void set_union(DISJ_SET S, set_type root1, set_type root2)
{
    if( S[root1] < S[root2] ) /* root2 is deeper set */
        S[root1] = root2;   /* make root2 new root */
    else{
        if( S[root2] == S[root1] ) /* same height */
            S[root1]--;
        S[root2] = root1; /* make root1 new root */
    }
}
```

6.5 Path Compression

```c
SetType Find(ElementType X, DisjSet S)
{
    if(S[X] <= 0)    return X;
    else return S[X] = Find(S[X], S);
}
```

```c
SetType Find(ElementType X, DisjSet S)
{   ElementType root, trail, lead;
    for(root = X; S[root] > 0; root = S[root]);  /* find the root */
    for(trail = X; trail != root; trail = lead){
       lead = S[trail];
       S[trail] = root;
    }  /* collapsing */
    return root;
}
```

5: ordinaty, union by size, union by height, union by size with path compression, union by rank(union by height with path compression).

6.6 Worst case for Union-by-rank and Path compression

---

7 Graph Algorithms

7.1 definitions

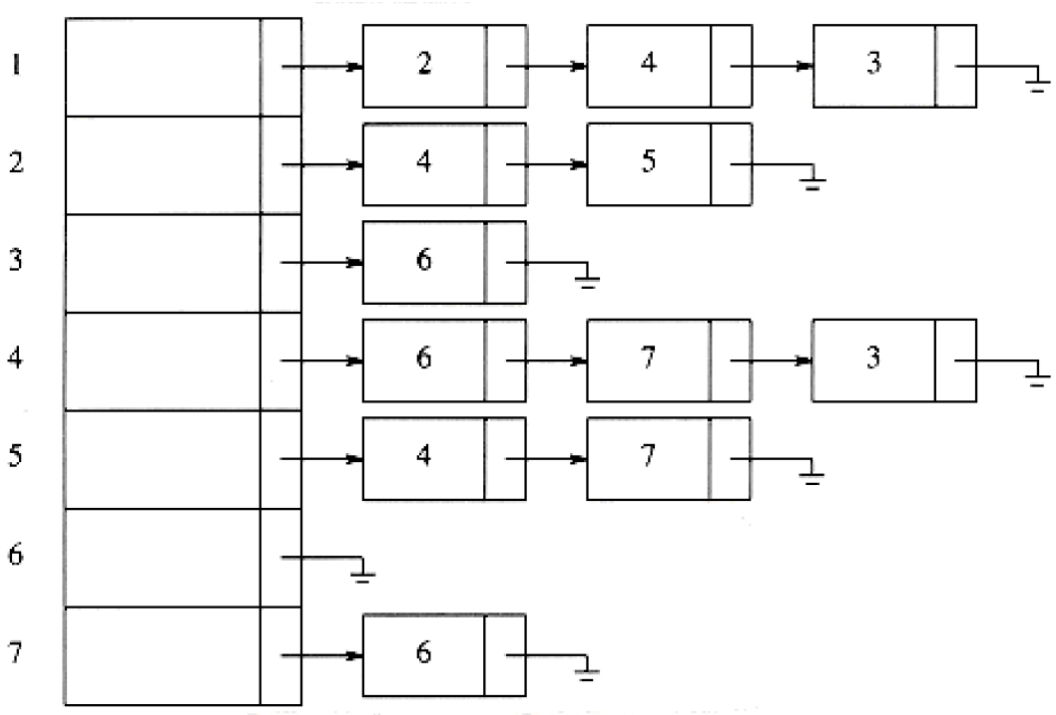
- G(V, E) : graph, vertices, edges
- Undirected graph :
  - $(v_i, v_j) = (v_j, v_i)$ ::= the same edge
  - v_i, v_j are *adjacent*
  - (v_i, v_j) is *incident on* v_i and v_j
- Directed graph (digraph) :
  - $< v_i, v_j > \neq < v_j, v_i >$
  - v_i is *adjacent to* v_j ; v_j is *adjacent from* v_i

  - $< v_i, v_j >$ is *incident on* v_i and v_j

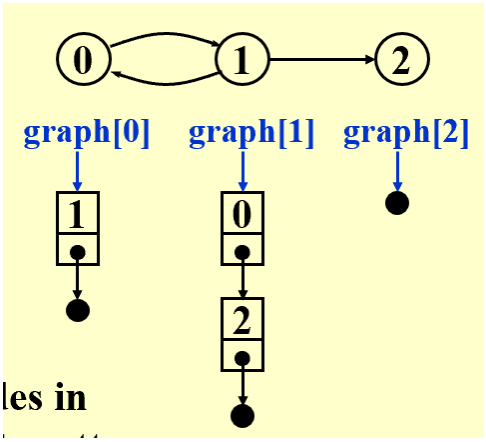
- Relations : self loop is illegal, and multigraph is not considered
- Complete graph : has the maximum number of edges
- Subgraph G' $\subset$ G : V(G') $\subseteq$ V(G) && E(G') $\subseteq$ E(G)
- Path ($\subset$ G) from v_p to v_q (a sequence of vertices)
- Length of a path : number of edges on the path
- Simple path : v_{i1}, v_{i2}, …, v_{in} are distinct
- Cycle : simple path with $v_p = v_q$
- (Connected) Component of an undirected G : the maximal connected subgraph
- A tree : a graph that is connected and acyclic (has no cycles)
- A DAG : a directed acyclic graph
- Strongly / Weakly connected directed graph G
- Strongly connected component
- Degree(v) : for a directed G, we have in-degree and out-degtee
- $e = (\sum_{i=0}^{n-1} d_i)/2$

Representation of Graphs

- Adjacency Matrix
  - If G is undirected, then adjacency matrix is symmetric, thus we can save space by storing only half of the matrix (as a 1-D array), and the index for a_{ij} is (i * (i - 1) / 2 + j)
  - $$degree(i) = \sum_{j=0}^{n-1} adj.mat[i][j] \quad if\ G\ is\ undirected$$ $$+ \sum_{j=0}^{n-1} adj.max[j][i] \quad if\ G\ is\ directed$$
  - T(N) = S(N) = $O(N^2)$
- Adjacency lists : replace each row by a linked list



For each vertex, we keep a list of all adjacent vertices. The space requirement is then O|E| + |V|). The leftmost structure is merely an array of header cells. If the edges have weights, then this additional information is also stored in the cells.



The order of nodes in each list does not matter

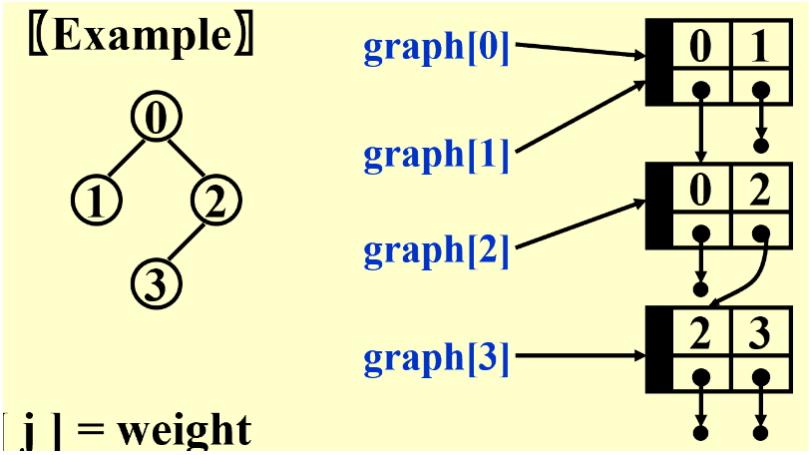
For undirected G :

- S = *n* heads + *2e* nodes = (*n*+*2e*)ptrs + *2e* ints (store twice.)
- Degree(i) = number of nodes in graph[i]
- T of examine E(G) = O(n+e)

For directed G :

- add inverse adjacency lists
- multilist representation

- Adjacency Multilists



- Weighted Edges

7.2 Topological Sort (拓扑排序)

AOV Network : V(G) represents activities and E(G) represents precedence ralations, and it must be a *dag (directed acyclic graph)*

- predecessor, immediate predecessor
- successor, immediate successor

Partial order : transitive and irreflexive

A **topological order** is a linear ordering of the vertices of a graph such that, for any two vertices, i, j, if i is a predecessor of j in the network then i precedes j in the linear ordering. (not be unique for a network)

```
void Topsort(Graph G)
{
    int Counter;
    Vertex V, W;
    for(Counter=0; Counter < NumVertex; Counter++){
        V = FindNewVertexOfDegreeZero(); /* O(|V|) */
        if(V == NotAVertex){
            Error("Graph has a cycle");
            break;
        }
        TopNum[V] = Counter; /* or output V */
        for(each W adjacent to V)
            Indegree[W]--;
    }
}
/* T = O(|V|^2) */
```

```
void Topsort(Graph G)
{
    Queue Q;
    int Counter = 0;
    Vertex V, W;
    Q = CreateQueue(NumVertex); MakeEmpty(Q);
    for(each vertex V)
        if(Indegree[V] == 0) Enqueue(V, Q);
    while(!IsEmpty(Q)){
        V = Dequeue(Q);
        TopNum[V] = ++Counter; /* assign next */
```

```
        for(each W adjacent to V)
            if(--Indegree[W] == 0) Enqueue(W, Q);
    } /* end while */
    if(Counter != NumVertex)
        Error("Graph has a cycle");
    DisposeQueue(Q); /* free memory */
}
/* T = O(|V|+|E|) */
```

7.3 Shortest Path Algorithms

Given a digraph G = (V, E), and a cost function c(e) for e ∈ E(G). The length of a path P from source to destination is $\sum_{e_i \in P} c(e_i)$ (also called **weighted path length**).

7.3.1 Single-Source Shortest-Path Problem

Given as input a weighted graph, G = (V, E), and a distinguished vertex, s, find the shortest weighted path from s to every other vertex in G.

Negative-cost cycle: no.
If there is no negative-cost cycle, the shortest path from s to s is defined to be zero.

7.3.1.1 Unweighted Shortest Paths:

Breadth-first search: queue

Table[i].Dist // initialized to be infty except for s
Table[i].Known // 1 if checked, 0 if not
Table[i].Path // initialized to be 0

```
void Unweighted(Table T)
{
    int CurrDist;
    Vertex V, W;
    for(CurrDist=0; CurrDist < NumVertex; CurrDist++)
    {
        for(each vertex V)
            if(!T[V].known && T[V].Dist == CurrDist){
                T[V].known = true;
                for(each W adjacent to V)
                    if(T[W].Dist == Infinity){
                        T[W].Dist = CurrDist + 1;
                        T[W].path = V;
                    } /* end-if Dist == Infinity */
            } /* end-if !known && Dist == CurrDist */
    } /* end-for CurrDist */
}
/* without queue, T = O(|V|^2)
```

```
void Unweighted(Table T)
{ /* T is initialized with the source vertex S given */
    Queue Q;
    Vertex V, W;
    Q = CreateQueue(NumVertex); MakeEmpty(Q);
    Enqueue(S, Q); /* Enqueue the source vertex */
    while(!IsEmpty(Q)){
        V = Dequeue(Q);
```

```
        T[V].known = true; /* not really necessary */
        for(each W adjacent to V)
            if(T[W].Dist == Infinity){
                T[W].Dist = T[V].Dist + 1;
                T[W].Path = V;
                Enqueue(W, Q);
            }
    }
    DisposeQueue(Q);
}
/* T = O(|V|+|E|) */
```

7.3.1.2 Dijkstra's Algorithm (weighted) (⭐) 。

贪心算法，但是能找到全局最优解

Let S = { s and v_i's whose shortest paths have been found }

For any u ∈ S, **define distance [u] = minimal length of path { s →(vi ∈ S)→ u }.**

If the paths are generated in non-decreasing order, then

- the shortest path must go through ONLY $v_i \in S$;
- u is chosen so that distance[u] = min{$w_i \notin S$ | distance[w]} *Greedy Method*
- if distance[u_1] < distance[u_2] and we add u_1 into S, then distance [u_2] may change. If so, a shorter path from S to u_2 must go through u_1.

```
void Dijkstra(Table T)
{ /* T is initialized by Figure 9.30 on p.303 */
    Vertex V, W;
    for(;;){    /* O(|V|) */
        V = smallest unknown distance vertex;
        if(V == NotAVertex)
            break;
        T[V].Known = true;
        for(each W adjacent to V)
            if(!T[W].Known)
                if(T[V].Dist + Cvw < T[W].Dist){
                    Decrease(T[W].Dist to T[V].Dist + Cvw);
                    T[W].Path = V;
                }
    }
} /* not work for edge with negative cost */
```

7.3.1.3 Graphs with Negative Edge Costs

```
void WeightedNegative(Table T)
{ /* T is initialized by Figure 9.30 on p.303 */
    Queue Q;
    Vertex V, W;
    Q = CreateQueue(NumVertex); MakeEmpty(Q);
    Enqueue(S, Q); /*Enqueue the source vertex */
    while(!IsEnpty(Q)){ /*each vertex can dequeue at most |V| times */
        V = Dequeue(Q);
        for(each W adjacent to V)
```

```
            if(T[V].Dist + Cvw < T[W].Dist){ /* no longer once per edge */
                T[W].Dist = T[V].Dist + Cvw;
                T[W].Path = V;
                if(W is not already in Q)
                    Enqueue(W, Q);
            }
    }
    DisposeQueue(Q); /* free memory */
} /* negative-cost cycle will cause indefinite loop */
/* T = O(|V|x|E|)
```

7.3.1.4 Acyclic Graphs

If the graph is acyclic, vertices may be selected in topological order

T = O(|E| + |V|) and no priority queue is needed.

- Application: AOE (Activity On Edge) Networks
  - EC[j] \ LC[j] ::= the earlist \ latest completion time
  - Lasting time \ Slack time
  - CPM (Critical Path Method)

7.3.2 All-pairs Shhortest Path Problem

For all pairs of v_i and v_j ($i \neq j$), find the shortest path between.

- Method 1: Use single-source algorithm for |V| times. T = $O(|V|^3)$ - works fast on sparse graph
- Method 2: $O(|V|^3)$ algorithm given in Ch.10, works faster on dense graphs.

7.4 Network Flow Problems

7.4.1 A simple algorithm

Find any path $s \rightarrow t$ in G_r : Flow G_f (augmenting path)

Take the minimum edge on this path as the amount of flow and add to G_f

Update G_r and remove the 0 flow edges : Residual G_r

shortcoming: we cannot be greedy at this point, so it's wrong.

7.4.2 A solution – allow the algorithm to undo its decision

For each edge (v, w) with flow f_v, w in G_f , **add an edge (w, v)** with flow f_v, w in G_r

If the edge capabilities are **rational numbers**, this algorithm always terminate with a maximum flow. (The algorithm works for G with cycles as well.)

7.4.3 Analysis (If the capacities are all integers)

- An augmenting path can be found by an unweighted shortest path algorithm.
  $T = O(f \cdot |E|)$ where f is the maximum flow.

- Always choose the augmenting path that allows the largest increase in flow
  /* modify Dijkstra's algorithm */
  $T = T_{augmenting} * T_{find\_a\_path} = O(|E| \log cap_{max}) * O(|E| \log |V|)$
  $= O(|E|^2 \log |V|)$ if cap_{max} is a small integer

- Always choose the augmenting path that has the least number of edges
  /* unweighted shortest path algorithm */
  $T = T_{augmenting} * T_{find\_a\_path} = O(|E|) * O(|E||V|)$

The **min-cost flow** problem is to find, among all maximum flows, the one flow of minimum cost provided that each edge has a cost per unit of flow.

7.5 Minimum Spanning Tree

A **spanning tree** of a graph G is a tree which consists of V(G) and a subset of E(G)

- The minimum spanning tree is a tree since it is acyclic : n(edges) = |V| – 1.
- It is minimum for the total cost of edges is minimized.
- It is spanning because it covers every vertex.
- A minimum spanning tree exists iff G is connected.
- Adding a non-tree edge to a spanning tree, we obtain a cycle.

7.5.1 Prim's Algorithm - grow a tree

greedy method and similar to Dijkstra's algorithm

adding a vertex into seleted part, with minimum weight edges between this vertex and the selected part.

7.5.2 Kruskal's Algorithm - maintain a forest

```
1  void Kruskal(Graph G)
2  {
3      T = {};
4      while(T contains less than |V|-1 edges && E is not empty){
5          choose a least cost edge(v,w) from E; /*DeleteMin */
6          delete(v,w) from E;
7          if((v,w) does not create a cycle in T)
8              add(v,w) to T; /* Union / Find */
9          else discard(v,w);
10     }
11     if(T contains fewer than |V|-1 edges)
12         Error("No spanning tree");
13 }
14 /* T = O(|E|log|E|)
```

7.6 Application of Depth-Fist Search

```
/* a generalization of preorder traversal */
void DFS(Vertex V)
{
    visited[V] = true;
    for(each W adjacent to V)
        if(!visited[W]) DFS(W);
}
/* T = O(|E|+|V|) as long as adjacency lists are used */
```

7.6.1 Undirected Graphs

```
void ListComponents(Graph G)
{
    for(each V in G)
        if(!visited[V]){
            DFS(V);
            printf("\n");
        }
}
```

7.6.2 Biconnectivity

G is a `biconnected graph` if G is connected and has **no articulation points.**

A `biconnected component` is a maximal biconnected subgraph.

Finding the biconnected components of a connected undirected G (⭐):

- Use the depth first search to obtain a spanning tree of G (`Back edges`)
- Find the `articulation points` (关节点) in G
  - The root is an articulation point iff it has at least 2 children
  - Any other vertex u is an articulation point iff u has at least 1 child, and it is impossible to move down at least 1 step and then jump up to u's ancestor

$$Low(u) = \min[\ Num(u), \min Low(w), \min Num(b)\ ]$$
$$w\ is\ a\ child\ of\ u,\ and\ (u,b)\ is\ a\ back\ edge.$$

u is an **articulation point** iff
(1) u is the **root** and has **at least 2 children**; or
(2) u is not the root, and has **at least 1 child** such that **Low(child) $\geq$ Num(u).**

`Tarjan` algorithm

7.6.3 Euler Circuits

Euler tour : Draw each line exactly once without lifting your pen from the paper

> An Euler tour is possible if there are **exactly two** vertices having odd degree. One must start at **one of the odd-degree** vertices.

Euler Circuits : Draw each line exactly once without lifting your pen from the paper, AND finish at the starting point

An Euler circuit is possible iff the graph is **connected** and each vertex has an **even degree.**

---

8 Sorting

8.1 Preliminaries

```
void X_SORT(ElementType A[], int N)
/* N must be a legal integer */
/* Assume integer array for the sake of simplicity */
/* '>' and '<' are the only operations allowed on the input data */
/* Consider internal sorting only (entire sort can be done in main memory)*/
```

8.2 Insertion Sort

```
void InsertionSort(ElementType A[], int N)
{
    int j,P;
    ElementType Tmp;

    for(P=1; P<N; P++){
        Tmp = A[P]; /* the next coming card */
        for(j=P; j>0 && A[j-1]>Tmp; j--)
            A[j] = A[j-1];
        A[j] = Tmp;
    }
}
/* The worst case: T(N) = O(N^2)
   The best case: T(N) = O(N) */
```

8.3 A Lower Bound for Simple Sorting Algorithms

inversion (逆序对)

Swapping two adjacent elements that are out of place removes exactly one inversion.

T(N, I) = O(I + N), where I is the number of inversions in the original array.

The average number of inversions in an array of N distince numbers is N(N-1)/4
So any algorithm that sorts by exchanging adjacent elements $T_{average} = \Omega(N^2)$.

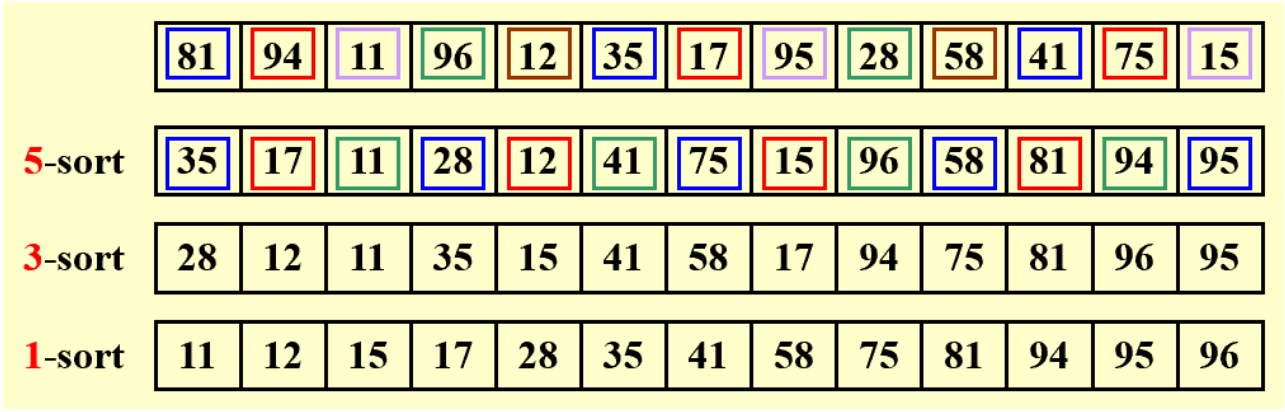
8.4 Shellsort – by Donald Shell

- Shellsort is a very simple algorithm, yet with an extremely complex analysis. It is good for sorting up to moderately large input (tens of thousands).

Define an *increment sequence* $h_1 < h_2 < \cdots < h_t (h_1 = 1)$

Define an h_k*-sort* at each phase for k = t, t-1, …, 1

An h_k-sorted file that is then h_{k-1}-sorted remains h_k-sorted.



8.4.1 Shell's increment sequence: $h_t = [N/2]$, $h_k = [h_{k+1}/2]$

```
1  void Shellsort (ElementType A[], int N)
2  {
3      int i, j, increment;
4      ElementType Tmp;
5      for(increment = N/2; increment > 0; increment /= 2)
6          /* h sequence */
7          for(i = increment; i < N; i++){
8              /* insertion sort */
9              Tmp = A[i];
10             for(j = i; j >= increment; j -= increment)
11                 if(Tmp < A[j - increment])
12                     A[j] = A[j - increment];
13                 else break;
14             A[j] = Tmp;
15         }
16 }
```

$T_{worst}(N) = O(N^2)$

8.4.2 Hibbard's Increment Sequence: $h_k = 2^k - 1$

$T_{worst}(N) = O(N^{3/2})$, $T_{avg}(N) = O(N^{5/4})$

8.4.3 Sedgewick's

Sedgewick's best sequence is $\{1, 5, 19, 41, 109, \ldots\}$ in which the terms are either of the form $9 \times 4^i - 9 \times 2^i + 1$ or $4^i - 3 \times 2^i + 1$

$T_{worst}(N) = O(N^{4/3})$, $T_{avg}(N) = O(N^{7/6})$

8.5 Heapsort

- Although Heapsort gives the best average time, in practice it is slower than a version of Shellsort that uses Sedgewick's increment sequence.

8.5.1 Algorithm 1

```
{
    BuildHeap(H); /* O(N) */
    for(i = 0; i < N; i++)
        TmpH[i] = DeleteMin(H); /* O(log N) */
    for(i = 0; i < N; i++)
        H[i] = TmpH[i]; /* O(1) */
}
```

T(N) = O(N log N), the space requirement is doubled.

8.5.2 Algorithm 2

```
void Heapsort(ElementType A[], int N)
{
    int i;
    for(i = N / 2; i >= 0; i--) /* BuildHeap */
        PercDown(A, i, N);
    for(i = N - 1; i > 0; i--){
        Swap( &A[0], &A[i]); /* DeleteMax */
        PercDown(A, 0, i);
    }
}
```

The average number of comparisons used to heapsort a random permutation of N distinct items is 2N log N - O(N log log N) .

Although Heapsort gives the best average time, in practice it is slower than a version of Shellsort that uses Sedgewick's increment sequence.

8.6 Mergesort

- Mergesort requires linear extra memory, and copying an array is slow. It is hardly ever used for internal sorting, but is quite useful for **external sorting.**

```
void MSort(ElementType A[], ElementType TmpArray[], int Left, int Right)
{
    int Center;
    if(Left < Right){
        /* if there are elements to be sorted */
    Center = (Left + Right) / 2;
    MSort(A, TmpArray, Left, Center);   /* T(N/2) */
    MSort(A, TmpArray, Center + 1, Right);  /* T(N/2) */
    Merge(A, TmpArray, Left, Center + 1, Right);/*O(N)*/
    }
}

void Mergesort(ElementType A[], int N)
{
    ElementType *TmpArray;  /* need O(N) extra space */
    TmpArray = malloc(N * sizeof( ElementType));
    if(TmpArray != NULL){
    MSort(A, TmpArray, 0, N - 1);
    free(TmpArray);
    }
    else FatalError("No space for tmp array!!!");
}
```

```
/* Lpos = start of left half, Rpos = start of right half */
void Merge( ElementType A[ ], ElementType TmpArray[ ],
           int Lpos, int Rpos, int RightEnd )
{   int  i, LeftEnd, NumElements, TmpPos;
    LeftEnd = Rpos - 1;
    TmpPos = Lpos;
    NumElements = RightEnd - Lpos + 1;
    while( Lpos <= LeftEnd && Rpos <= RightEnd ) /* main loop */
        if ( A[ Lpos ] <= A[ Rpos ] )
    TmpArray[ TmpPos++ ] = A[ Lpos++ ];
        else
    TmpArray[ TmpPos++ ] = A[ Rpos++ ];
    while( Lpos <= LeftEnd ) /* Copy rest of first half */
        TmpArray[ TmpPos++ ] = A[ Lpos++ ];
    while( Rpos <= RightEnd ) /* Copy rest of second half */
        TmpArray[ TmpPos++ ] = A[ Rpos++ ];
    for( i = 0; i < NumElements; i++, RightEnd - - )
         /* Copy TmpArray back */
        A[ RightEnd ] = TmpArray[ RightEnd ];
}
```

T(N) = O(N + N log N)

8.7 Quicksort (⭐)

the fastest known sorting algorithm in practice

8.7.1 The Algorithm

```
void Quicksort ( ElementType A[ ], int N )
{
     if ( N < 2 )  return;
     pivot = pick any element in A[ ];
     Partition S = {A[] \ pivot} into two disjoint sets:
       A1={a in S|a in pivot}and A2={a in S|a in pivot};
     A = Quicksort(A1,N1) or {pivot} or Quicksort(A2,N2);
}
/* The best case T(N) = O(N logN)
```

8.7.2 Picking the Pivot

- a wrong way: Pivot = A[0], T(N) = $O(N^2)$
- a safe maneuver: Pivot = random select from A[], but random number generation is expensive
- Median-of-Three partitioning

8.7.3 Partitioning Strategy

if there is a key == pivot: stop i and j both and then swap (there will be many dummy swaps, but at least the sequence will be partitioned into two equal-sized subsequences)

8.7.4 Small Arrays

Problem: quicksort is slower than insertion sort for small N (≤ 20)

Solution: cutoff when N gets small and use other efficient algorithm

8.7.5 Implementation

```
void Quicksort(ElementType A[], int N)
{
    Qsort( A, 0, N - 1 );
    /* A:   the array   */
    /* 0:   Left index  */
    /* N - 1: Right index   */
}
/* Return median of Left, Center, and Right */
/* Order these and hide the pivot */
ElementType Median3(ElementType A[],int Left,int Right)
{
    int  Center = ( Left + Right ) / 2;
    if ( A[ Left ] > A[ Center ] )
        Swap( &A[ Left ], &A[ Center ] );
    if ( A[ Left ] > A[ Right ] )
        Swap( &A[ Left ], &A[ Right ] );
    if ( A[ Center ] > A[ Right ] )
        Swap( &A[ Center ], &A[ Right ] );
    /* Invariant: A[Left] <= A[ Center ] <= A[ Right] */
    Swap(&A[Center], &A[Right - 1]); /* Hide pivot */
    /* only need to sort A[Left + 1] … A[Right - 2] */
    return  A[ Right - 1 ];  /* Return pivot */
}
void Qsort(ElementType A[], int Left, int Right)
{   int i, j;
    ElementType Pivot;
    if(Left + Cutoff <= Right){
        /* if the sequence is not too short */
      Pivot = Median3( A, Left, Right );
        /* select pivot */
      i = Left; j = Right - 1;
        /* why not set Left+1 and Right-2? */
      for( ; ; ){
        while(A[++i] < Pivot){ }/* scan from left */
        while(A[--j] > Pivot){ }/* scan from right */
        if ( i < j )
            Swap(&A[i], &A[j]); /* adjust partition */
        else break; /* partition done */
      }
      Swap(&A[i], &A[Right - 1]); /* restore pivot */
      Qsort(A, Left, i - 1);
      Qsort(A, i + 1, Right);
    }  /* end if - the sequence is long */
    else InsertionSort( A + Left, Right - Left + 1 );
}
```

8.7.6 Analysis

$$T(N) = T(i) + T(N-i-1) + cN$$

☞ **The Worst Case:**

$$T(N) = T(N-1) + cN \Rightarrow T(N) = O(N^2)$$

☞ **The Best Case: [... ...] • [... ...]**

$$T(N) = 2T(N/2) + cN \Rightarrow T(N) = O(N \log N)$$

☞ **The Average Case:**

Assume the average value of $T(i)$ for any i is $\frac{1}{N}\left[\sum_{j=0}^{N-1} T(j)\right]$

$$T(N) = \frac{2}{N}\left[\sum_{j=0}^{N-1} T(j)\right] + cN \Rightarrow T(N) = O(N \log N)$$

〖Example〗 Given a list of N elements and an integer k. Find the kth largest element.

8.8 Sorting Large Structures

Problem: swapping large structures can be very much expensive

Solution: add a pointer field to the structure and swap pointers instead - **indirect sorting**. Physically rearrange the structures at last if it's really necessary

(Example) Table sort: the sorted list - list[table[0]], list[table[1]],…, list[table[n-1]]

8.9 A General Lower Bound for Sorting

【Theorem】 Any algorithm that sorts by comparisons only must have a worst case computing time of **Ω(N log N).**

8.10 Bucket Sort and Radix Sort

8.10.1 Bucket Sort

(Example) Suppose that we have N students, each has a grade record in the range 0 to 100 (thus there are M = 101 possible distinct grades). How to sort them according to their grades in linear time?

```
Algorithm
{
    initialize count[];
    while(read in a student's record)
          insert to list count[stdnt.grade];
    for(i = 0; i < M; i++){
        if(count[i]) output list count[i];
    }
}
/* T(N, M) = O(M + N) */
```

8.10.2 Radix Sort (基数排序)

(Example) Given N = 10 integers in the range 0 to 999 (M = 1000) Is it possible to sort them in linear time?
Input: 64, 8, 216, 512, 27, 729, 0, 1, 343, 125, Sort according to the **Least Significant Digit** first.

Bucket	0	1	2	3	4	5	6	7	8	9
Pass 1	0	1	512	343	64	125	216	27	8	729
Pass 2	0	512	125		343		64			
	1	216	27							
	8		729							
Pass 3	0	125	216	343		512		729		
	1									
	8									
	27									
	64									

Output: 0, 1, 8, 27, 64, 125, 216, 343, 512, 729

T=O(P(N+B)) where P is the number of passes, N is the number of elements to sort, and B is the number of buckets.

…后面笔记一点没做，看这个吧.jpg

https://zhang-each.github.io/My-CS-Notebook/DS/ds

- 选择、堆、快速排序可以在每次运行后找到排序列表中至少一个元素的最终位置
-