Chapter 20 Introduction to Algorithms & Data Structures

An efficient algorithm needs $\begin{cases} \text{correct data structure (how data is stored)} \\ \text{abstraction} \end{cases}$

■ Big-Oh notation

↳ considers the asymptotic behavior and ignores constant factors

↓ what happens for large input size only.

Definition :

$$f(x) \text{ is } O(g(x)) \Leftrightarrow \exists\, x_0, c.\ \forall x > x_0,\ f(x) \leq c * g(x).$$

i.e., $f(x)$ grows slower than $g(x)$.

$\lim_{x\to\infty} \left\| \frac{f(x)}{g(x)} \right\|$	$f(x)$ is $O(g(x))$?	$g(x)$ is $O(f(x))$
0	✓	✗
non-zero, finite	✓	✓
∞	✗	✓
undefined	✗	✗

$$O(1) < O(\lg N) < O(N) < O(N\lg N) < O(N^c) < O(2^N) < O(N!)$$

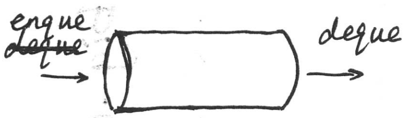
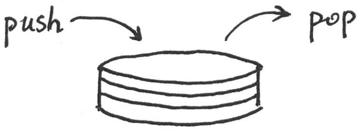
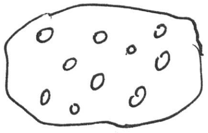
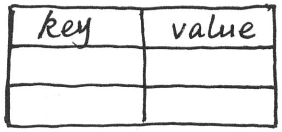
$\lg N$ ↓ logarithmic time

$N\lg N$ ↓ linearithmic time

→ intractable

- NP-complete problem : the best known algorithm for any of these problems requires exponential time. $O(2^N)$

A $O(N^c)$ solution to any NP complete problem can be transformed into a $O(N^c)$ solution to any OTHER NP-complete problems.

	Queues	Stacks	Sets	Maps
Concept	 First-in First-out (FIFO) sequence of items.	 Last-in First-out (LIFO) sequence of items	 a collection of elements	 key-value pairs
Application	process requests / data	Call stack; reverse sequence of items nested matching, e.g., parsing; undo + redo;	track ~~e~~ what items belongs to a group; check conflicts (union) in task scheduling; track which items in a ds we've worked on.	Social networking: UID-profile UID - {friends UID};

ADT	Queue<T>{
	enqueue (push);
	dequeue (pop)
	peek }
		Stack<T> {
		push;
		pop;
		peek} (top & pop)
			Set<T>{
			add; numItems;
			remove; contains;
			intersect; unionsets; }
				Map<K,V>{
				add (K,V); numItems;
				remove (K);
				lookup (K); }

Efficient Data Structure	Linked List vector / array	Linked list vector / array	BST (totally ordered keys) Hash table	BST Hash table

Chapter 21 Linked Lists

Linked list : a linear sequence of node , connected by pointers

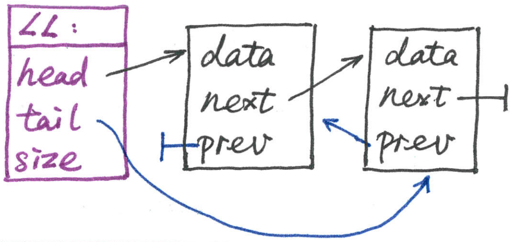
(only pointers to a type may be used in the declaration of the type)

singly linked list / doubly linked list

illustration

singly linked list:

doubly linked list:



Class declaration

singly linked list:

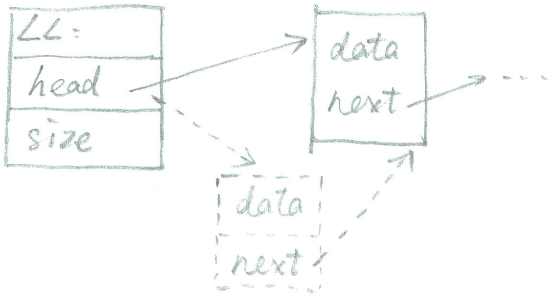
```
template < typename T >
class LinkedList {
    class Node {
        public :
            T data;
            Node * next;
    };      Node (_data, _next) : data
                    (_data), next(_next) { }

    Node * head ;
    size_t size ;

public :
    LinkedList ( ) : head ( NULL ),
                     size (0) { }
};
```

doubly linked list:

```
template < typename T >
class LinkedList {
    class Node {
        public :
            T data ;
            Node * next ;
            Node * prev ;
    };
    Node * head ;
    Node * tail ;
    Size_t size ;
public :
    LinkedList ( ) : head ( NULL ), tail (NULL),
                     Size (0) { }
};
```

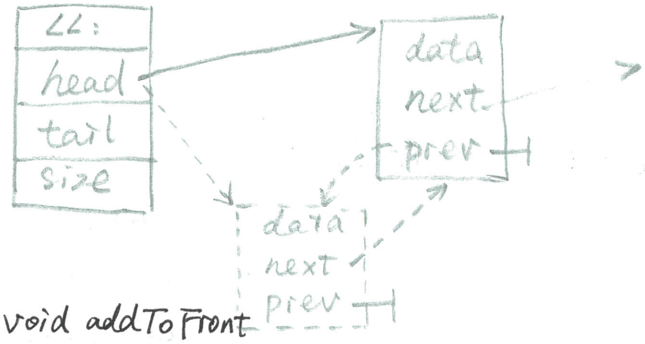
Add to Front

singly linked list:



```
void addToFront ( T data ) {
    head = new Node ( data , head );
    size ++ ;
}
```

doubly linked list:



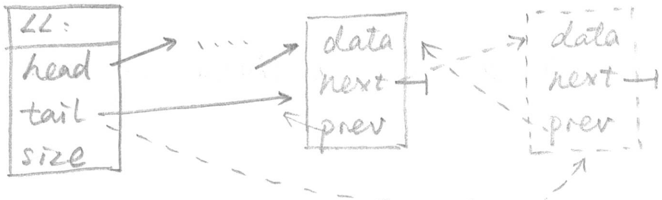
```
void addToFront
    head = new Node ( data , head ) ;
    if ( tail == NULL )    tail = head ;
    else    head -> next -> prev = head ;
    size ++ ;
}
```

Add to Back

singly linked list

```
void addToBack ( T data ){
    Node * tail = head ;
    while ( tail != NULL ){
        tail = tail-> next ;
    }
    tail = new Node ( data, NULL);
    size++ ;
}
```

doubly linked list



```
void addToBack ( T data ){
    tail = new Node ( data, NULL, tail );
    if ( head == NULL )
        head = tail;
    else
        tail-> prev-> next = tail ;
    size ++ ;
}
```

Copy constructor

singly linked list

```
LinkedList ( const LinkedList & rhs ) :
head ( NULL ), size( rhs. size ) {
    if ( rhs. head != NULL ){
        Node * it = rhs. head->next;
        head = new Node ( rhs. head ->
                    data , NULL );
        Node * tail = head ;
        while ( it != NULL){
            tail->next = new Node
                (it->data , NULL);
            tail = tail->next ;
            it = it->next ;
        }
    }
}
```

doubly linked list

①

```
LinkedList ( const T& rhs) : head(NULL), size(0){
    Node * it = rhs. tail ;
    while ( it != NULL ){
        addToFront ( it-> data ) ;
        it = it-> prev;
    }
}
```

②

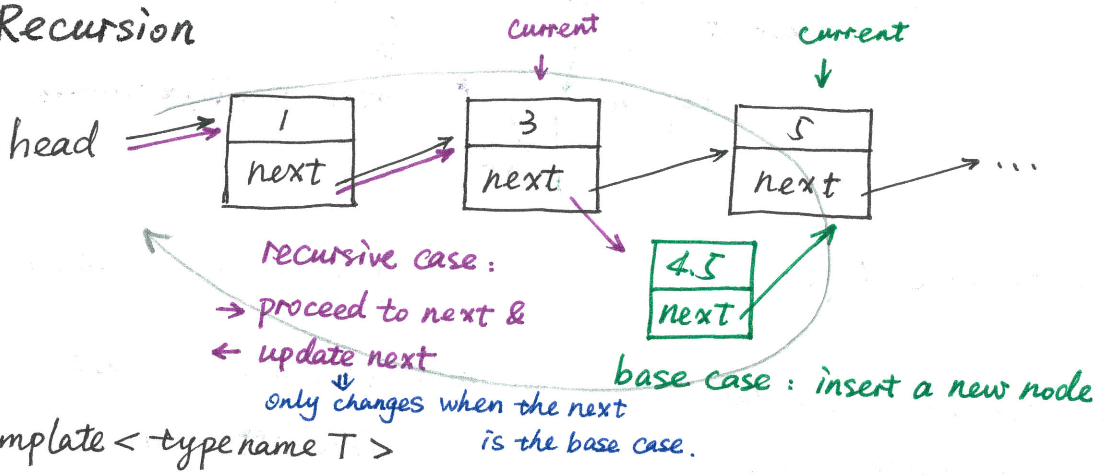
```
{
    Node * it = rhs. head ;
    while ( it != NULL) {
        addToBack ( it->data );
        it = it-> next;
    }
}
```

destructor

```
~LinkedList ( ){
    while ( head != NULL ){
        Node * tmp = head ;
        head = head-> next ;
        delete tmp ;
    }
}
```

■ Insert in sorted order

1. Recursion



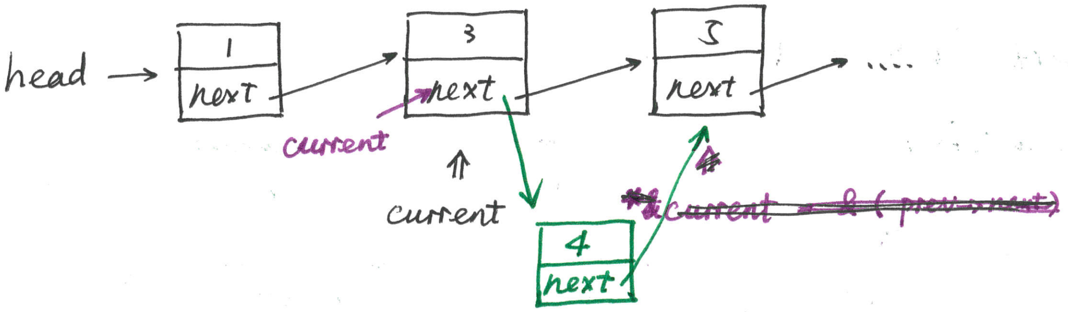
recursive case :
→ proceed to next &
← update next
⇓
only changes when the next is the base case.

base case : insert a new node

```
template < typename T >
class LinkedList {
          ⋮
    private :
        class Node { } ;
        // helper method
        Node * addSorted ( const T& data , Node * current ) {
            if ( current == NULL || data < current -> data ) {
                return new Node ( data , current ) ;
            }
            current -> next = addSorted ( data , current -> next ) ;
            return current ;
        }
    public :
        void addSorted ( const T& data ) {
            head = addSorted ( data , head ) ;
        }   size++ ;
};
```

2. A pointer to a pointer

↳ an elegant algorithm instead of a more staight forward iterative method, where (& head) and (& next) both can be represented by (current)



Pointer to node before

```
void addSorted ( const T& data ){
    if ( head== NULL || data <
    head-> data ) {
        head= new Node ( data, head);
    }
    else {
        Node * current = head;
        while ( current -> next != NULL
        && data > current->next->data){
            current = current-> next ;
        }
        current -> next = new Node (
            data, current-> next ) ;
    }
}
```

A pointer to a pointer

```
void addSorted ( T data ){
    Node ** current = & head ;
    while ( * current != NULL &&
            data > (* current)-> data){
        current = & (* current)->next ;
    }
(next) * current = new Node ( data, * current);
    size++ ;
    // if doubly linked
    if( (* current)-> next == NULL ){
        (* current)-> prev = tail ;
        tail = * current ;
    }
    else{
        (* current)-> prev = (* current)
                    -> next-> prev;
        (* current)-> next -> prev = * current;
    }
}
```

Removing from a List

Singly Linked List

Doubly Linked List

- Remove from Front : O(1)

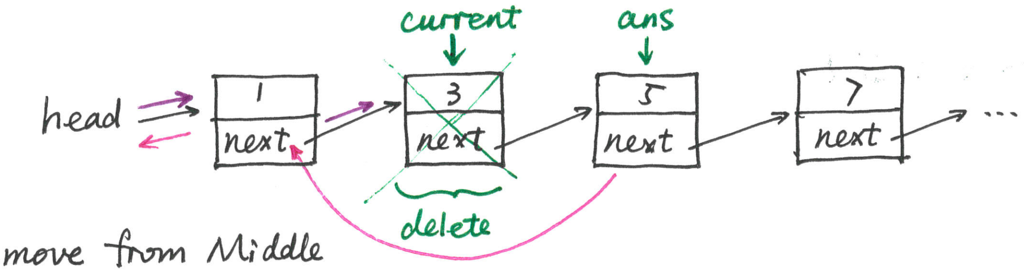
```
void RmFront(){
    if ( head != NULL){
        Node * tmp = head ;
        head = head-> next ;
        delete tmp ;
    }   size -- ;
}
```

```
void RmFront ( ){
    if ( head != NULL ){
        Node * tmp = head ;
        head = head-> next ;
        delete tmp ;
        size -- ;
        if ( head== NULL ){
            tail= NULL ;
        }
        else {
            head-> prev = NULL ;
        }
    }
}
```

- Remove from Back : O(N) or O(1)

```
void RmBack ( ){
    if ( head != NULL){
        Node ** curr = & head ;
        while ( (* curr)-> next != NULL)
            curr= & (* curr)->next ;
        delete ( * curr) ;
        * curr = NULL ;
        size -- ;
    }
}
```

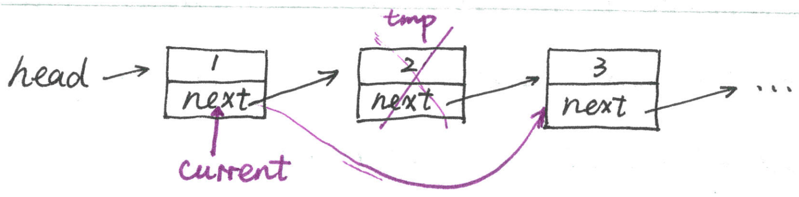
```
void RmBack ( ){
    if ( tail != NULL ){
        Node * tmp = tail ;
        tail = tail -> prev ;
        delete tmp ;
        size -- ;
        if ( tail == NULL ){
            head = NULL ;
        }
        else {
            tail -> next = NULL ;
        }
    }
}
```



- Remove from Middle

```
template<typename T>
class LinkedList {
        ⋮
    private:
        ⋮
        // helper method
        Node * remove ( const T & data , Node * current ) {
            // data not found ( including empty list )
            if ( current == NULL )
                return NULL ;
            // base case : node to remove
            if ( data == current -> data ) {
                Node * ans = current -> next ; // ans -> prev = current -> prev ;
                delete current ;  // size -- ;
                return ans ;
            }
            // recursive case
            current -> next = remove ( data , current -> next ) ;
            // if ( current -> next == NULL )   tail = current ;
            return current ;
        }
    public :
        void remove ( const T & data ) {
            head = remove ( data , head ) ;
            ~~size -- ;~~
        }
```

- Remove from Middle : using Pointer-to-a-pointer



```
void Remove ( const T & data ) {
    ~~if ( head != NULL ) {~~
        Node ** current = & head ;
        while ( (* curr) != NULL && data != (* curr) -> data ) {
            curr = & (* curr) -> next ;
        }
        if ( * curr != NULL ) {
            Node * tmp = * curr~~ent~~ ;
            * current = tmp -> next ;
            // if doubly linked
            /* if ( * curr == NULL ) {
                   tail = tmp -> prev ;
               }
               else {
                   (* curr) -> prev = tmp -> prev;
               }
            */
            delete tmp ; // size-- ;
        }
}
```

- Remove All Occurance

↳ Only one change compare to recursive remove (data) : call removeAll method on current -> next that continues to remove for the rest of the list, instead of stopping and returning.

```
template < typename T >
class LinkedList {
        ⋮
    private :
        template < bool removeAll >
        Node * remove ( const T & data , Node * current ) {
                if ( current = NULL )
                        return NULL ;
                if ( data == current-> data ) {
                        Node * ans ;
                        if ( removeAll ) {
                                ans = remove < removeAll > ( data , current-> next ) ;
                        }
                        else {
                                ans = current -> next ;
                        }
                        delete current ;
                        return ans ;
                }

                current-> next = remove < removeAll > ( data , current -> next ) ;
                return current ;
        }
public :
        void remove ( const T & data ) { head = remove < false > ( data, head ) ; }
        void removeAll ( const T & data ) { head = remove < true > ( data, head ) ; }
};
```

Chapter 22 Binary Search Trees (BST)

■ Concepts

1. Terminology:

- Graph : a collection of nodes and edges.
- Tree : a data structure comprised of nodes and edges which is a connected graph that has no undirected cycles

  ⇓ (at least one undirected path between any two nodes)

  ⇓ (an undirected path which forms a "loop")
- Rooted tree : a tree in which one particular node is the root node (there exists directed path from it to every others)
- Binary tree : a rooted tree where each node has at most two outgoing edges.
- Binary Search Tree : a binary tree which has one invariant.

  ✪ everything to the left < that given node < everything to the right

Properties:

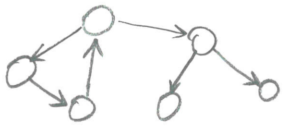
- Depth (of a node) : the length of the path from the root to itself

  (root has depth 0)
- Height (of a node) : maximum length of path from it to a leaf node

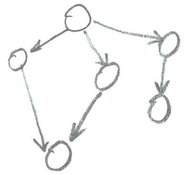
  (leaf node has height 1)
- Height (of a tree) : the height of its root
- Full : every node either has 0 or 2 children.
- Balanced : for every node in the tree, the heights of its children differ by at most 1.
- Complete : Every level (except possibly the last), has as many

nodes as it possibly can have. The last level is filled in from left to right.

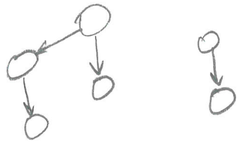
Example 1: Graphs not trees



has a directed cycle

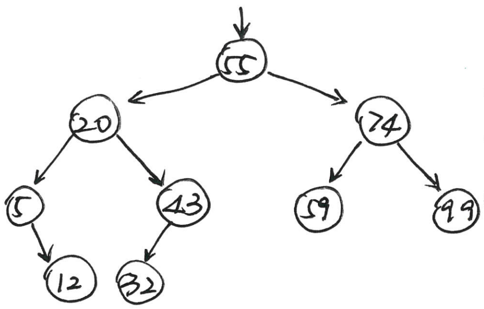


has undirected cycle

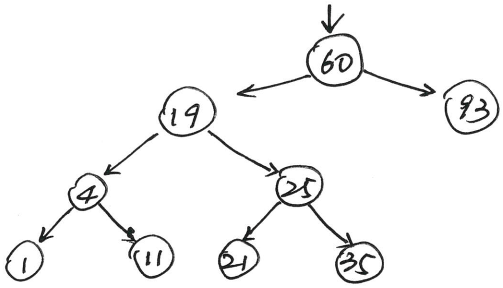


forest

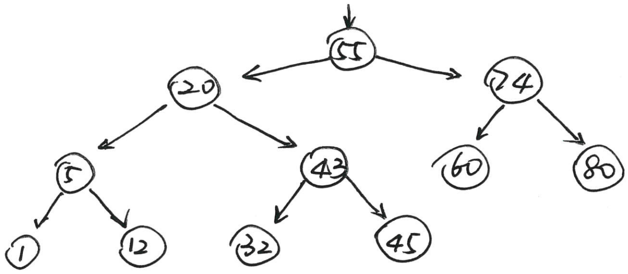
Example 2: BSTs with different properties



balanced ✓
full X
complete X



balanced X
full ✓
complete X



balanced ✓
full X
complete ✓

2. Applications:

① Implement maps & sets with $O(\lg N)$ for addition, lookup, removal. for totally ordered type (where we can compare any two elements)

② Resource management
which finds the smallest key greater than or equal to a particular value.

3. Other useful tree: abstract syntax trees for parsing input/code.

■ Adding to a BST

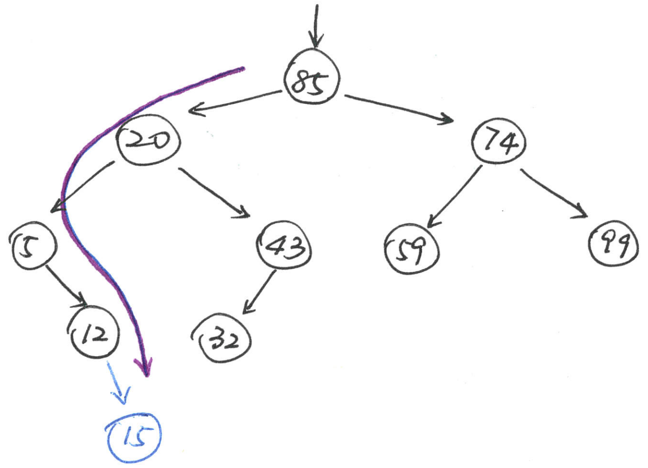
idea : ① The newly added node should be leaf node.

② The correct place is found by going from root to NULL

smaller → left, larger → right.

1. Recursion :

```
Node * addNode ( Node * curr, const K & key ) {
    if ( curr == NULL )
        return new Node ( key );
    else {
        if ( key < curr-> key ) {
            // recursively goes down
            curr -> left = addNode ( curr->left , key );
        }
        else {
            if ( key > curr-> key )
                curr-> right = addNode ( curr-> right , key );
        }
    }
    return curr ;
}

root = addNode ( root, key );
```



2. Pointer to a Pointer to a Node.

```
void addNode ( const K & key ){
    Node ** curr = & root ;
    while ( * curr != NULL ){
        if ( key < (*curr) -> key )
            curr = &(*curr) -> left ;
        ~~else{~~
        else if ( key > (* curr) -> key )
                curr = & ( * curr) -> right ;
        }
        else {
            return ;
        }
    }
    * curr = new Node( key ) ;
}
```

■ Search a BST

Iterative :

```
bool search ( const K & key ) { const {
    const Node * curr = root ;
    while ( curr != NULL ) {
        if ( key == curr->key )
            return true ;
        else if ( key < curr-> key )
            curr = curr-> left ;
        else
            curr = curr -> right ;
    }
    return false ;
```

■ Remove from a BST

idea : ① if toRm has 1 or 0 child , then delete it directly .

② otherwise : find the most similar node that has 0 or 1 child ,
put its data into the node toRm , then rm that node .

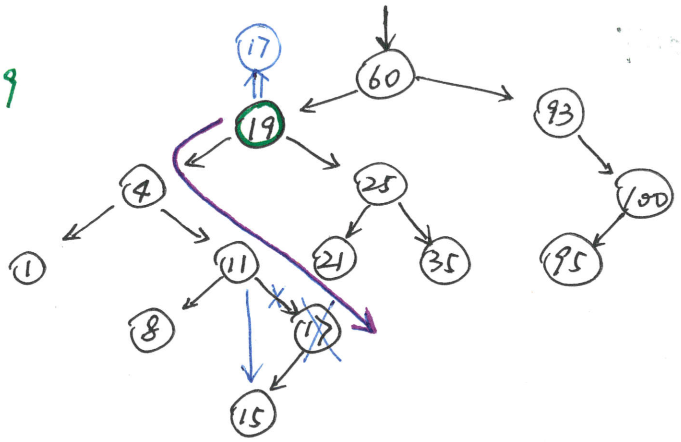
(i) go left , then all the way to the right , or

(ii) go right , then all the way to the left .

1. Recursion

```
Node * removeNode ( Node * curr, const K & key ) {
    if ( curr == NULL )
        return curr ;
    if ( key < curr-> key )
        curr → left = removeNode ( curr->left, key ) ;
    else if ( key > curr-> key )
        curr -> right = removeNode ( curr-> right, key ) ;
    else {
        if ( curr-> left == NULL ) {
            Node * tmp = curr-> right ;
            delete curr ;
            return tmp ;
        }
        else if ( curr -> right == NULL ) {
            Node * tmp = curr-> left ;
            delete curr ;
            return tmp ;
        }
        else {
            Node * tmp = maxValueNode ( curr-> left ) ;
            curr-> key = tmp-> key ;
            curr -> left = remove Node ( curr->left , tmp-> key ) ;
        }
    }
    return curr ; }
```

delete 19



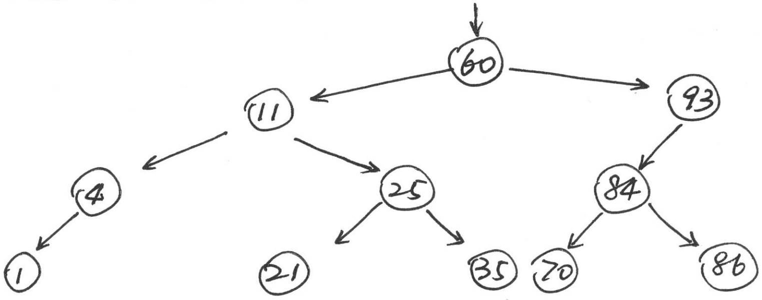
```
Node *  maxValueNode ( Node * curr) {
      while ( curr-> right != NULL)
            curr =  curr -> right ;
      return curr;
}

void remove ( const K& key) {
      root = removeNode ( root, key);
}
```

2. A Pointer to a Pointer to a Node :

```
void remove ( const K & key) {
    Node ** curr = &root ;
    while ( * curr != NULL && (*curr) -> key != key ) {
        if ( key < (* curr) -> key )
            curr = (* curr) -> left ;
        else
            curr = (* curr) -> right ;
    }
    if ( * curr != NULL) {
        if ( (*curr)-> left == NULL ) {
            Node * tmp = * curr ;
            * curr = tmp-> right ;
            delete tmp ;
        }
```

```
        else if ( (*curr)->right == NULL ){
            Node * tmp = *curr;
            * curr = tmp->left;
            delete tmp;
        }
        else {
            Node ** toRp = &(*curr)->left;
            while ( (*toRp)->right != NULL )
                toRp = &(*toRp)->right;
            (*curr)->key = (*toRp)->key;
            Node * tmp = *toRp;
            *toRp = tmp->left;
            delete tmp;
        }
    }
}
```

Tree Traversals



1. Inorder : 1 4 11 21 25 35 60 70 84 86 93.

```
void printInorder ( Node * curr ) {
    if ( curr != NULL ) {
        printInorder ( curr->left );
        std::cout << curr->key << " ";
        printInorder ( curr->right );
    }
}
```

2. Preorder : 60 11 4 1 25 21 35 93 84 70 86

Add the items to an empty tree using this order will reconstruct the tree with exactly the same structure.

```
void printPreorder ( Node * curr) {     // duplicate Tree (   ) {
    if ( curr != NULL ) {
        Std:: cout << curr-> key << " " ;    // add ( curr->key);
        print Preorder ( curr-> left);       // duplicate ( - - - )
        print Preorder ( curr-> right );
    }
}
```

3. Postorder : 1 4 21 35 25 11 70 86 84 93 60

Used to destroy a tree.

```
void destroy ( Node * curr) {
    if ( curr != NULL ) {
        destroy ( curr->left ) ;
        destroy ( curr-> right ) ;
        delete curr;
    }
}
```

4. Reverse

swap the order of acting on the left first to acting on the right first.

BST example 1 :

Determines if BT has a continuous sequence of items along a path which sum to the specified target.

```
bool hasPathSum ( const Node * curr , int currSum, int target ) {
    if ( target ==  currSum )
        return true;
    if ( curr == NULL )
        return false;
    return (  hasPathSum ( curr->left , curr->sum + curr->data , target ) ||
                "   "   "   "  " ->right,      "  "  "  - " "    |        ||
              has Path Sum ( curr->left , curr->data , target ) ||
                "   "   "   "     ->right ,    "  "  "  "  "   ) ;
}
bool has Path Sum ( int target ) {
      return hasPathSum ( root , 0 , target );
}
```

BST example 2 :

Check BT obeys BST rules. Given minInTree & maxInTree.

ans1 :

```
bool isBSTOrdered ( Node * curr, const T& min, const T& max ) {
    if ( curr == NULL ) {
        return true;
    }
    if ( curr->data < min || curr->data > max ) {
        return false;
    }
```

```
    return isBSTOrdered (curr->left,   min,   curr->data) &&
           isBSTOrdered (curr->righ, curr->data, max )) ;
}

bool isBSTOrdered () {
    if ( root ~~!~~== NULL )
        return true;
    return isBSTOrdered ( root, minInTree(), maxInTree() ) ;
}
```

Chapter 23 Hash Tables

Store data in an array / ~~list~~ vector by array[index] = key.

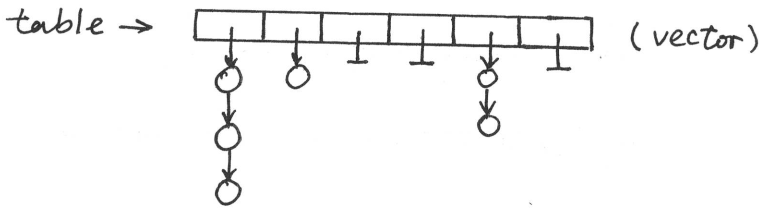
The index is determined by: index = hash(key) % num_buckets.

$$\text{key} \xrightarrow{\text{hashing function}} \underset{\equiv\ hash(key)}{\text{unsigned int}\ (H)} \xrightarrow{H\%\underset{\downarrow \atop \text{number of buckets}}{N}} \text{index}$$

■ Collision Resolution

↳ more than one data is stored in the same bucket.

1. Chaining.



2. Open Addressing : seeking a nearby index which is not used.

(1) linear probing : step size is constant.

(2) quadratic probing : increment the step each time.

problem :

If it supports removing, then we should distinguish between a "truly empty" bucket and the one ~~what~~ which had data but deleted.

After "truly empty" buckets disappear, the table needs to be cleaned-up (or periodically clean-up).

■ Hashing Functions

1. Criteria for a "good" hashing function.

(1) valid : ① purely a function of its input

② $\forall\ a=b$, $hash(a) == hash(b)$

(2) very likely to get different hash values for objects that we consider different.

2. Basic designing principles

(1) should incorporate all parts of the object.

(2) combine the data in ways that permutations and alternations create different results.

(3) test it to see if it's actually good.

3. Crytographic Hash Functions

↳ for securety - sensive purposes

(1) Application : e.g., storage of login information.

(2) Approach : store the salt and hash (password + salt)

↓

randomly generated string

Otherwise, it's easy to be attacked by dictionary attack and brute force attacks in parallel.

↓

store [words , hash(words)]

■ Rehashing

↳ resize the table as the number of elements in it grows.

1. When : Load factor ($\frac{num\_data}{num\_bucket}$) should in 0.5 ~ 0.8 range.

2. How : ① Allocate a larger array / vector twice as large (or next prime number in a pre-prepared array of primes)

② Iteratively take all items from the old table, rehash and place them into the ~~ne~~ new table.

Chapter 24 Heaps & Priority Queues

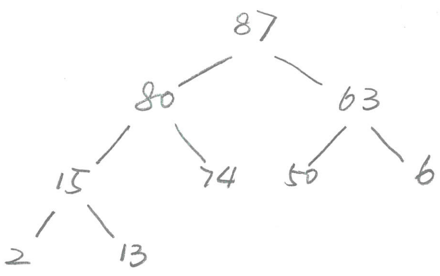
priority queue : a queue where each item has an associated priority, and the next item returned from the queue is the one with the highest priority (a lower number)

Heap : a data structure which gives efficient access to its largest (or smallest) element.

- conceptually ~ complete binary tree that obeys the heap's ordering rule.
- actually ~ array

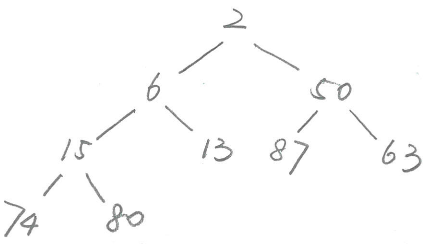
■ Heap Concept

max heap



parent > children

min heap

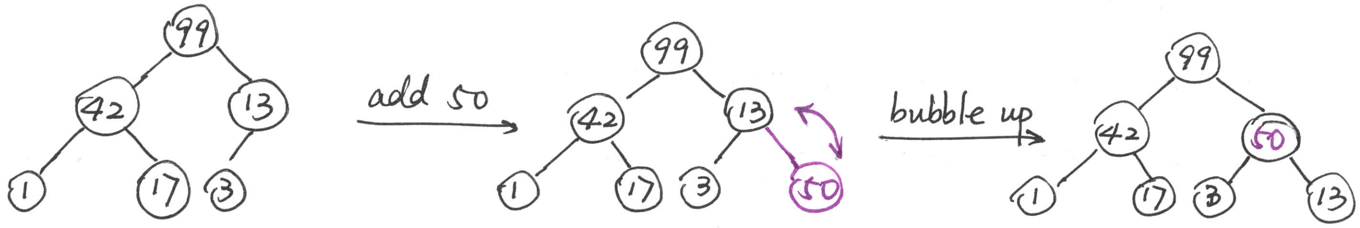


parent < children

1. Insertion : insert at the next available place

   ↓

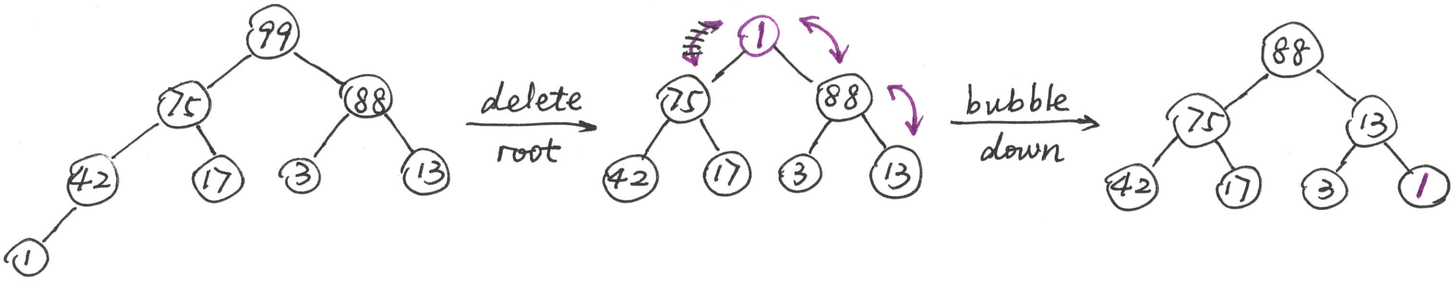
   recursively "bubble up" to fix the ordering



2. Deletion : swap the very last item with the root

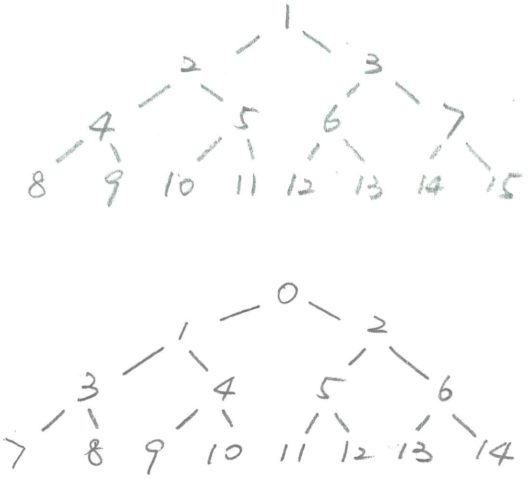
↓

recursively "bubble down" to fix the ordering

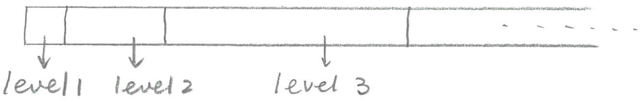


■ Heap Array Implementation

	Root at 0	Root at 1
parent	(i-1)/2	i/2
Left child	2*i+1	2*i
right child	2*i+2	2*i+1



If root is at array[1], then array[0] is **sentinel** : a special item that is not actually part of the heap's data, but is ordered so so that it stops the bubble up process without a special case.



e.g., for int, we can use INT_MIN or INT_MAX in #include<limit.h>.

■ Priority Queue

1. STL's Priority Queue :

std:: priority_queue < Value, Container, Compare >

default : Container = vector<T>

Compare = less<T> // max heap

2. Application : Compression — Huffman Coding

(1) fundation :

① The frequency of symbols in the input will typically not be uniform.

② We could do better if we encode the more common characters with fewer bits at the expense of encoding the less common characters with more bits.

(2) Algorithm to find the optimal encoding where no symbol's encoding is a prefix of another.

① leaf nodes ~ input symbols

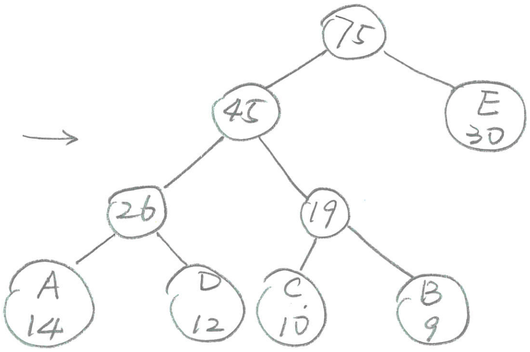
② encoding ~ the path from the root to the leaf,

go left = 0, go right = 1

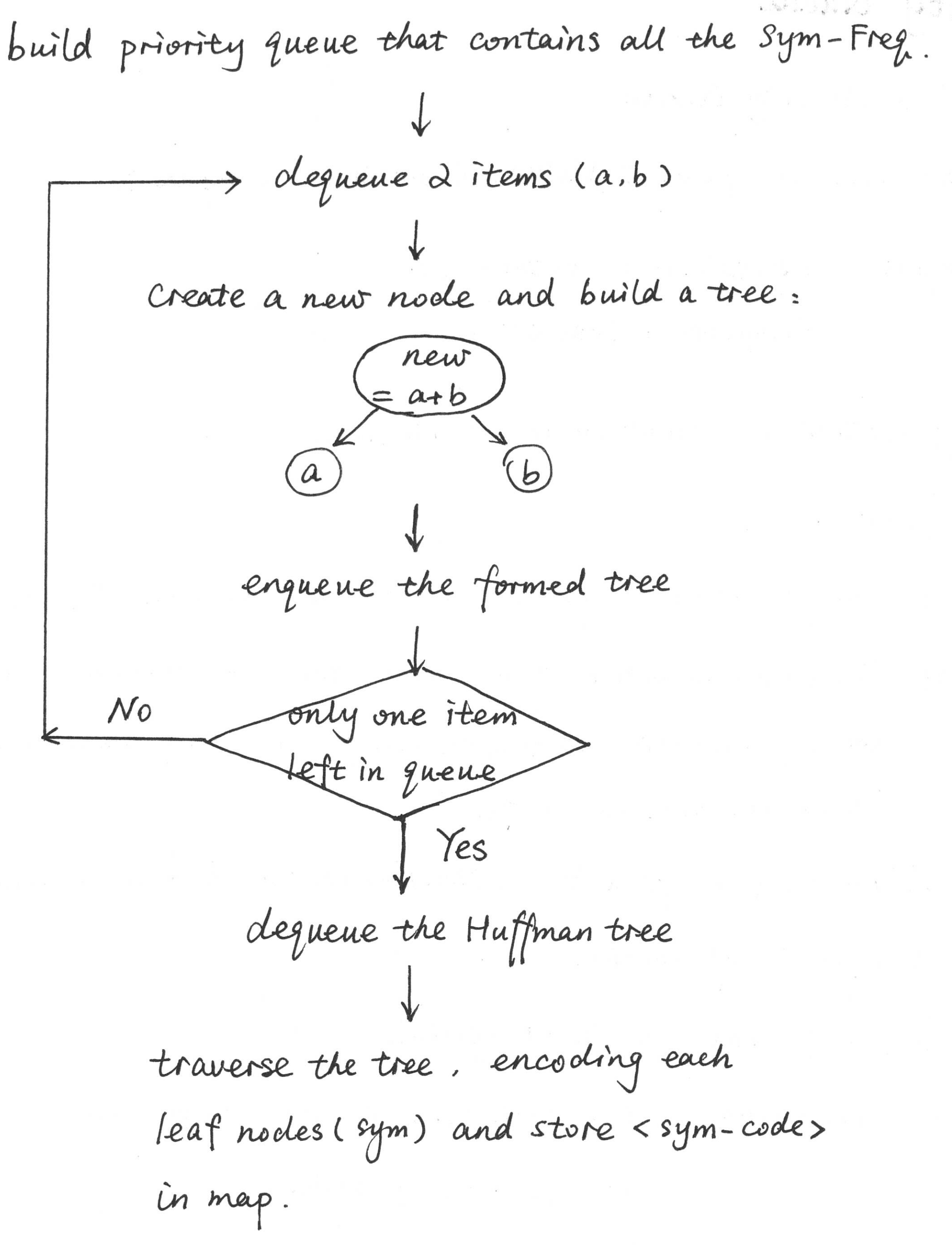
③ Store the encoding as map.

Example:

Sym	Freq
A	14
B	9
C	10
D	12
E	30



Sym	Encoding
A	000
B	011
C	010
D	001
E	1


build priority queue that contains all the Sym-Freq.
dequeue 2 items (a,b)
Create a new node and build a tree:
new = a+b
a
b
enqueue the formed tree
only one item left in queue
No
Yes
dequeue the Huffman tree
traverse the tree, encoding each leaf nodes (sym) and store < sym-code > in map.

	Array	Linked List	BST	Hash Table	Heap
Storage space	O(N)	O(N)	O(N)	O(2N) = O(N)	O(N)
Insertion	O(1) or O(N)	O(1)	O(lgN)	O(1)	O(lgN)
Sorted insertion	O(N)	O(N)	O(lgN)	N/A	N/A
Deletion	O(N)	O(1)	O(lgN)	O(1)	O(lgN)
Random access	O(1)	O(N)	O(lgN)	O(1)	N/A
min/max access	O(1) if sorted	O(1) if sorted	O(lgN)	N/A	O(1)
Traversal	O(N)	O(N)	O(N)	N/A	O(N)
Application	Stack, queue	stack, queue	~~priori~~ Maps Sets, parsing	maps, sets	priority queue

Chapter 25 Graphs

Applications

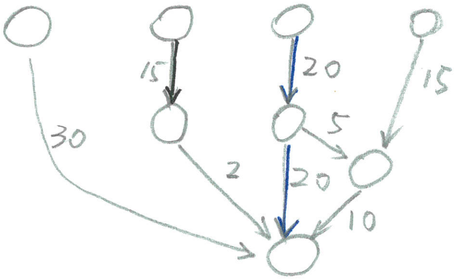
1. Task Scheduling

- Goal : complete the tasks all in the smallest time possible
- Scheduling graph : DAG (directed acyclic graph)

  ↳ contains no directed cycles ; undirected is permitted

- critical path constaints how quickly we can complete the entire work.

  Tasks not on the critical path have some slack.



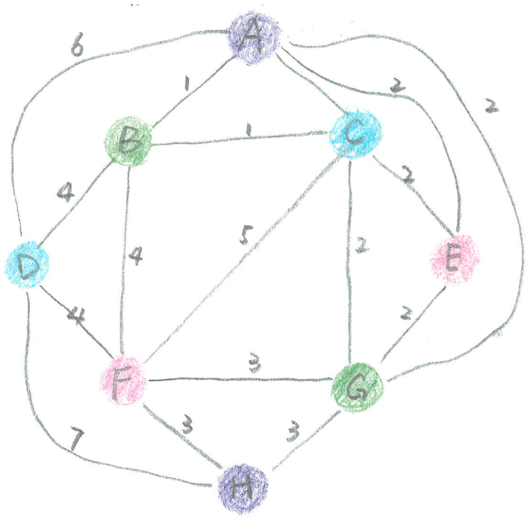
20 + 20 : critical path

2. Resource Allocation

- Goal : determine an assignment of users to resources such that there are no conflicts
- graph : interference graph (a graph in which two nodes are connected by an edge if they conflict with each other)
- Algorithm : graph coloring
  - assign colors to each node (user) such that no two adjacent nodes have the same color (a particular resource)
  - Upper limit : Four Color Theorem for planar graph
  - Efficiency { NP - complete , for best answer

    polynomial approximations , for reasonably good answer.

	1	2	3	4	5	6	7
A	x	x				x	
B	x			x		x	
C	x	x			x		
D				x		x	x
E		x					
F			x	x	x		
G		x	x				
H			x				x



3. Path Planning

- Goal: find a path (shortest, or other requirements) from one location to another.
- Algorithm: Dijkstra

4. Social Networks

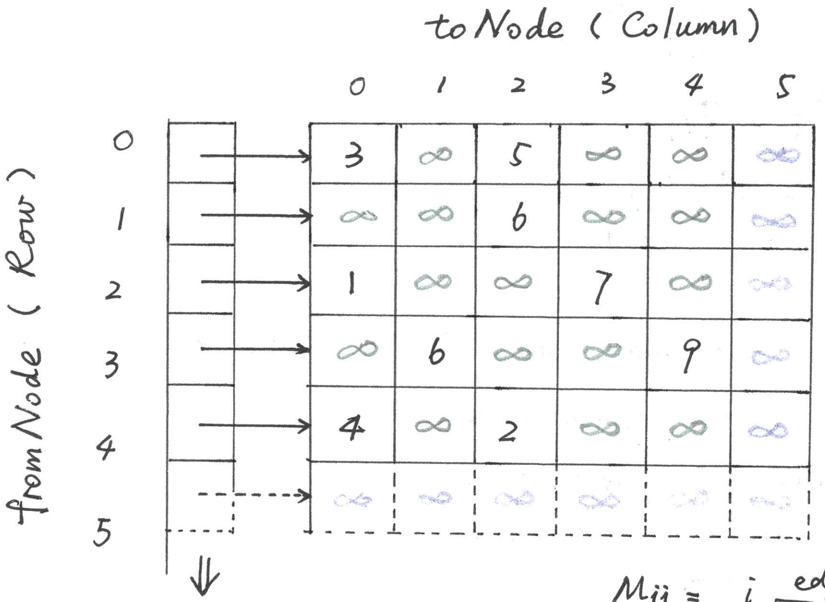
- Goal: provide features to the users, enhance advertising revenue.

■ Graph Implementations

```
                        node              weight
                       ↗                    ↑
template < typename N , typename E , bool directed >
class Graph {
    void addNode ( const ~~&~~ N & nodeInfo ) ;
    void removeNode ( const N & nodeInfo ) ;
    void addEdge ( const N & fromNode , const N& toNode , const E& edgeInfo ) ;
    void removeEdge ( const N& fromNode , const N& toNode ) ;
    set < const N& > getNodes () const ;
    set < pair< const N& , E & > > getAdjacencies ( const N& which Node ) ;
            // consider only one direction .
```

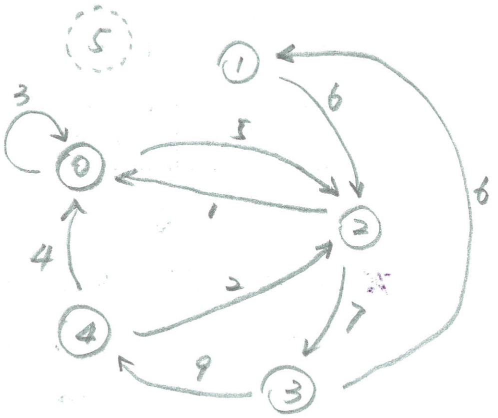
```
    const E & getEdge ( const N& fromNode , const N& toNode ) const;
    bool isAdjacent ( const N& fromNode , const N& toNode ) const;
};
```

1. Adjacency Matrix



toNode (Column)

fromNode (Row)	0	1	2	3	4	5
0	3	∞	5	∞	∞	∞
1	∞	∞	6	∞	∞	∞
2	1	∞	∞	7	∞	∞
3	∞	6	∞	∞	9	∞
4	4	∞	2	∞	∞	∞
5	∞	∞	∞	∞	∞	∞



$M_{ij} = i \xrightarrow{edge} j$

★ { Vectors of vectors.
  | 2D array.

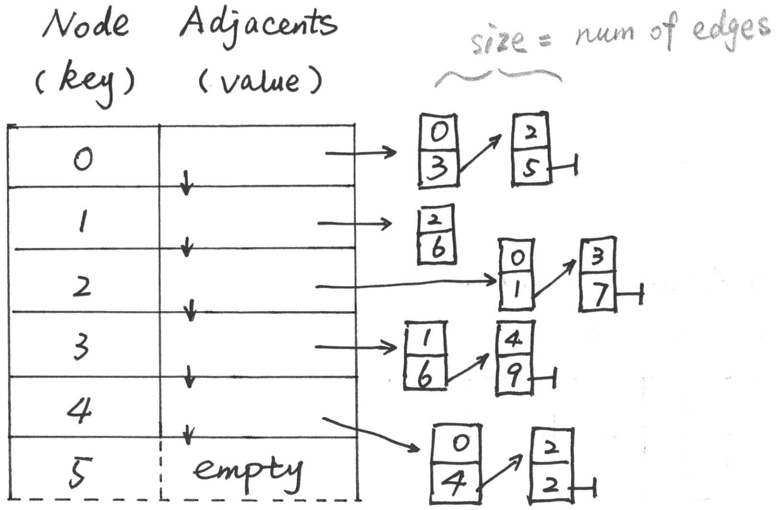
- Representation of " no edge "

  { Generic way ⊣ [keep one matrix of bools
                  [hold pointers to Es , use NULL for no edge.
  | Specific type : pick a particular value

- Representation of nodes

  { N is an unsigned int : N = index.
  | Otherwise : keep a map from Ns to unsigned ints

2. Adjacency List



⇓

★ map < N, map<N,E> > : Both maps can be implemented as the following.

→ Linked list < pair<N,E> >

→ Balanced ~~tree~~ BST < pair<N,E> >

→ Hash table < pair<N,E> >

~~→ Map < N,E >~~

Comparison :

	Adjacency Matrix	Adjacency List
add Node	• add a new row (vector) • add a column (expand each existing vector by 1)	• add a new <node, empty> pair to the main map
remove Node	• rm the vector • for all existing vectors, rm that element	• locate and rm the node from map • for all existing <N, E> pair maps, search for that N and rm it.
add Edge ; remove Edge	• modify matrix element Mij (modify Mji as well if undirected)	• add a new <Nj,Eij> to the specific map i. or • search in the specific map i for Node j and remove it

	Adjacency Matrix	Adjacency List
getNodes	• Traverse through that main vector, create a set of all nodes.	• Traverse through the main map, create a set of all nodes
getAdjacencies	• Traverse through the specific row, create a set of all ~~nei~~ neighbors <N, E>s.	~~• Traverse through~~ • locate that Node i. • Traverse through that map & create a set of all neighbors
getEdge ; isAdjacent	• visit the matrix element M_{ij}	• locate fromNode and search toNode in that map

Efficiency :

	Adjacency Matrix	Adjacency List with...		
		Linked List	Balanced BST	Hash Table
space	V^2	$V + E$ (total number of vertices/edges)		
addNode	V	1	$\lg V$	1
removeNode	V^2	V^2	$V \cdot \lg V$	V
addEdge removeEdge	1	V	$\lg V$	1
getNodes	V	V	V	V
getAdjacencies	V	V	$\lg V + \lvert ans \rvert$	$\lvert ans \rvert$
getEdge isAdjacent	1	V	$\lg V$	1
	The constant factors are likely much better	search : $O(N)$ remove : $O(1)$ add : $O(1)$	search : $O(\lg N)$ remove : $O(\lg N)$ add : $O(\lg N)$	search : $O(1)$ remove : $O(1)$ add : $O(1)$

- ## Algorithms

I. Graph Searches

1. Depth-first search (DFS)

A. Recursively

```
searchForDes ( node, visited ) {
        // base case
        if ( node == destination )
                return node ;
        // shrink size
        if ( node is in visited )
                return none ;
        visited . add ( node ) ;
        for next in getAdjacencies ( node ) {
                if ( searchForDes ( next ) != none )
                        return path ( node + next ) ;
                }
        }
        return none ;
}
```

B. Explicit Stack.

↳ part of worklist algorithms (keep a list of items to work on, take out an item → process it → Done)

↑ generate & add new item ↲

```
template < typename Worklist >
search ( origin, dest ) {
        Worklist todoList ; // DFS -> Stack, BFS -> Queue
        Set visited ;
        todoList. push ( path [ origin ] ) ;
        while ( ! todoList. empty() ) {
                currPath = todoList. pop() ;
                currNode = currPath. lastNode() ;
                if ( currNode == dest ) {
                        return currPath ;
                }
                if ( currNode $\notin$ visited ) {
                        visited. add ( currNode ) ;
                        for x in getAdjacencies ( currNode ) {
                                todoList. push ( currPath. addNode (x) ) ;
                        }
                }
        }
        return none ;
}
```

DFS Features :

① Fully explore one path before exploring any other paths

② Answer might be a much longer path than shortest.

③ Applications : find Strongly Connected Components , Topological Sort .

2. Breadth-first search (BFS)

see last page for implementations

Features:

① Answer has the fewest number of "hops",
i.e., traverses the smallest number of other nodes.

3. Dijkstra's Shortest Path Algorithm

Features:

~~each node/edge. find adjacencies~~

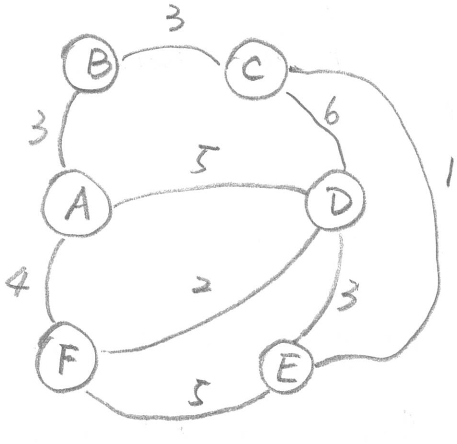
① Answer is the "shortest" path. ~~O(E·lgV)~~

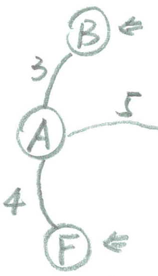
② Usage: no edges of negative weight
(otherwise, use Bellman-Ford algorithm)

Algorithm:

Create a list that stores shortest path from src to a particular node & the path length so far.

```
pick up src node, e.g., A
initialize the list with A-A ~0
      and others as ∞
while ( not all completed ) {
      curr = shortest path by far
             && uncompleted.
      mark curr as completed.
      for each neighbor next :
            if curr.length() + ~~next~~ getEdge(curr, next) < next.length
            then update the next.length() and next.path()
}
```

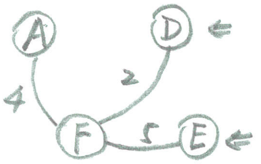




A-F-D = 6

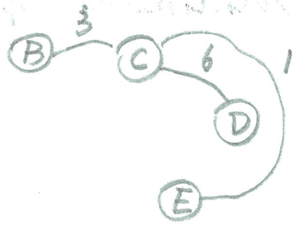
A-D = 5

∴ no update

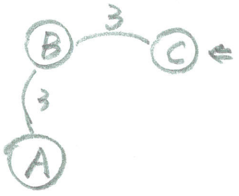


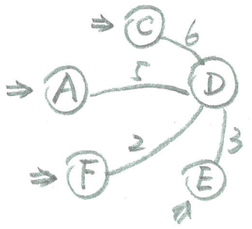
A-B-C - E = 6+1=7

A-D-E = 8 ∴ update E



							FINAL
src → A	0 (A-A)						0 (A-A)
B	∞	3 (A-B)					3 (A-B)
C	∞	∞	6 (A-B-C)	6 (A-B-C)	6 (A-B-C)		6 (A-B-C)
D	∞	5 (A-D)	5 (A-D)	5 (A-D)			5 (A-D)
E	∞	∞	∞	9 (A-F-E)	8 (A-D-E)	7 (A-B-C-E)	7 (A-B-C-E)
F	∞	4 (A-F)	4 (A-F)				4 (A-F)





A-D→E = 5+3=8

A-F-E = 9

∴ update E

Implementation:

- workList : priority queue
- search : worklist algorithms.
- complexity : search & update each adjacency in worklist : $O(\lg V)$
  in each step, do so for every adjacency : $\frac{E}{V}$
  A total of V steps

$\Rightarrow O(E \cdot \lg V)$

Ⅱ. Minimum Spanning Trees (MST)

↳ a subset of the graph in which the edges connect all of the node together with the minimum sum of edge weights.

1. Prim's Algorithm

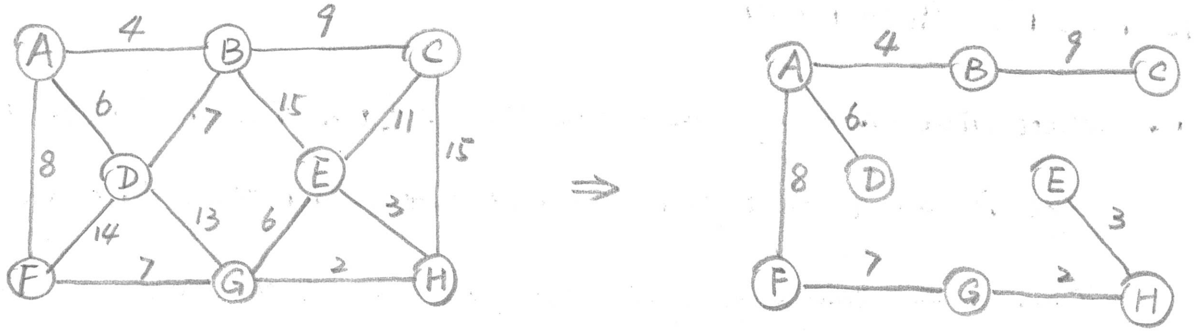
↳ maintains a connected tree at all times, and grows it one node at a time by picking the node which can be added with the lowest edge weight. "Greedy algorithm"

Implementation ①

priority queue :

edges	fromNode	toNode
(All edges	in the	graph)

```
for each x in getEdges ( A ) :
        PQ. push ( x ) ;
while ( MST. size() < Graph. size() ) {
        currEdge = PQ. pop() ;
        if ( currEdge. toNode( ) ∉ MST ) {
                MST. add ( currEdge ) ;
                for next in getAdjacencies ( currEdge. toNode( ) ) {
                        if ( next ∉ MST ) {
                                PQ. push ( edge [ curr. toNode -> next ] ) ;
                        }
                }
        }
}
```



Ordering of adding node :

Prim's: $A \xrightarrow{4} B$. $A \xrightarrow{6} D$. $A \xrightarrow{8} F$. $F \xrightarrow{7} G$, $G \xrightarrow{2} H$; $H \xrightarrow{3} E$. $B \xrightarrow{9} C$

Kruskal's: $G \xrightarrow{2} H$. $H \xrightarrow{3} E$. $A \xrightarrow{4} B$. $A \xrightarrow{6} D$. $F \xrightarrow{7} G$. $A \xrightarrow{8} F$. $B \xrightarrow{9} C$

Implementation ②

priority queue :

node	best-distance	connection
(All	vertices in the	graph)

```
for each x in getNodes() :
        PQ. push ( < x, ∞ > ) ;
while ( ! PQ. empty() ) {
        curr=PQ. pop() ;
        MST. add ( curr ) ;
        for next in getAdjacencies( curr.Node() )
                if ( next ∈ PQ && PQ. find (next). bestd ()
                    > getEdge ( curr.Node() -> next) ) {
                    update in PQ the next's best distance &
                                                  connection ;
                }
        }
}
```

2. Kruskal's Algorithm.

↳ start with ~~on~~ many one-node trees. Each time a lowest-weight edge is added that joins together two small trees into a larger tree. (i.e., that edge won't form a cycle in the MST — the two nodes have to belong to different trees)

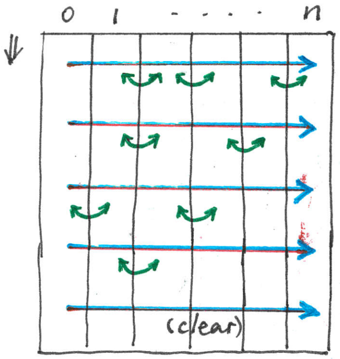
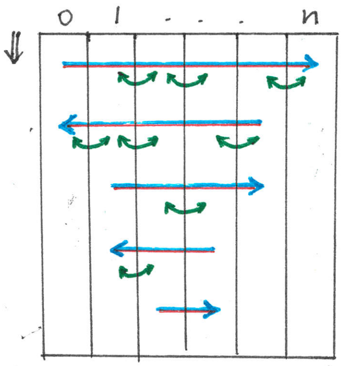
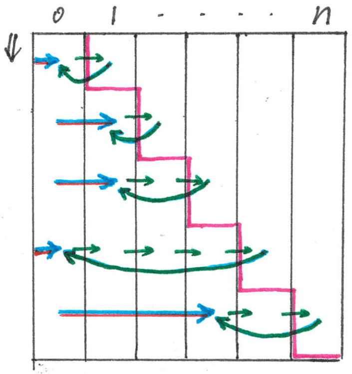
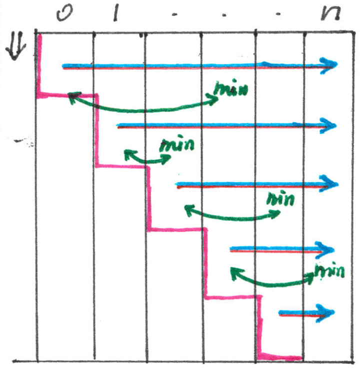
Implementation : union-find

III. Other algorithms

Problem	Description	Complexity
Clique + Independent set	A set of nodes which are all connected to each other / with no direct edges between any pair.	NP-complete
Isomorphism	$G1 \cong G2$ $\exists\ G2 = f(G1),\ \forall\ x \rightarrow y,\ f(x) \rightarrow f(y)$ $\forall\ x \not\rightarrow y,\ f(x) \not\rightarrow f(y)$	unknown.
Max Flow / Min Cut	src  sink, critical weight.	
Strongly Connected Components (SCC)	a set of nodes in a directed graph in which every node is reachable from every other node.	efficient with DFS
Topological Sort	DAG → ordering of nodes such that later ones "depend on" former ones.	
Traveling Salesperson Problem (TSP)	find minimum cost trip between a set of nodes	NP-complete

Chapter 26 Sorting

■ O(N²) for both array and linked list

	Bubble Sort	Shaker Sort	Insertion Sort	Selection Sort
Algorithm				

Code

Bubble Sort

```
int changed = 1;
while ( changed) {
  changed = 0;
  for(int i=0; i<n-1; i++){
    if( data[i+1]
       < data[i]){
      swap(& data[i],
        & data[i+1]);
      changed = 1;
    }
  }
}
```

Insertion Sort

```
int boundary = 1;
while (boundary < n){
  int curr = data[boundary];
  int pos = 0;
  while ( pos< boundary
          && curr > data[pos]){
    pos++;
  shift (data, pos, boundary);
  data[pos] = curr;
  boundary++;
}
```

Selection Sort

```
for( int pos=0; pos<n; pos++){
  int minIdx = findMin
        (data, pos, n);
  if ( minIdx != pos){
    swap ( data, minIdx,
        pos);
  }
}
```

	Bubble Sort	Shaker Sort	Insertion Sort	Selection Sort
Apply to Linked List	no less efficient on LLs than on arrays.		Very nice for LLs: build a new sorted LLs(reuse original nodes), and do sorted insertion each time.	no less efficient than on arrays.
	(swap data, not nodes)	only for doubly LLs.		
Feature	bubble elements much more quickly in the direction that we sort.		keep a sorted and unsorted region.	
				the sorted region is always the correct order

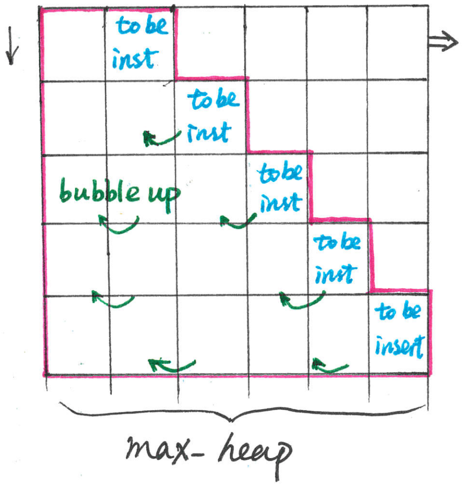
■ O(N* lgN)

1. Heap Sort

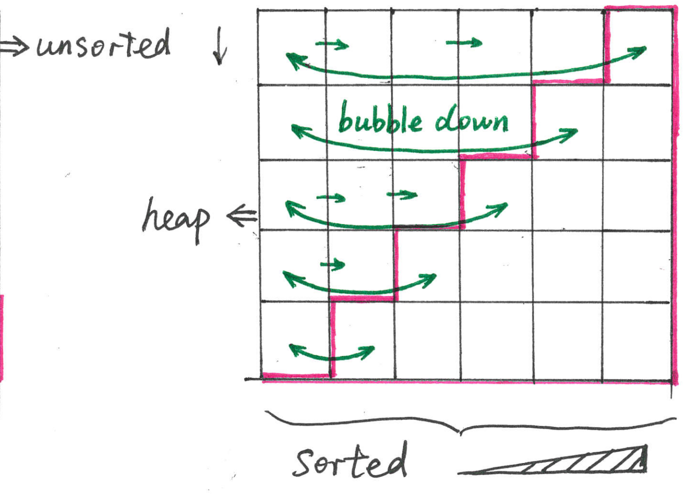
- idea : similar to insertion sort, but instead using heap to hold the sorted data.

- Algorithm :

① heapify :



② heap → sorted array.

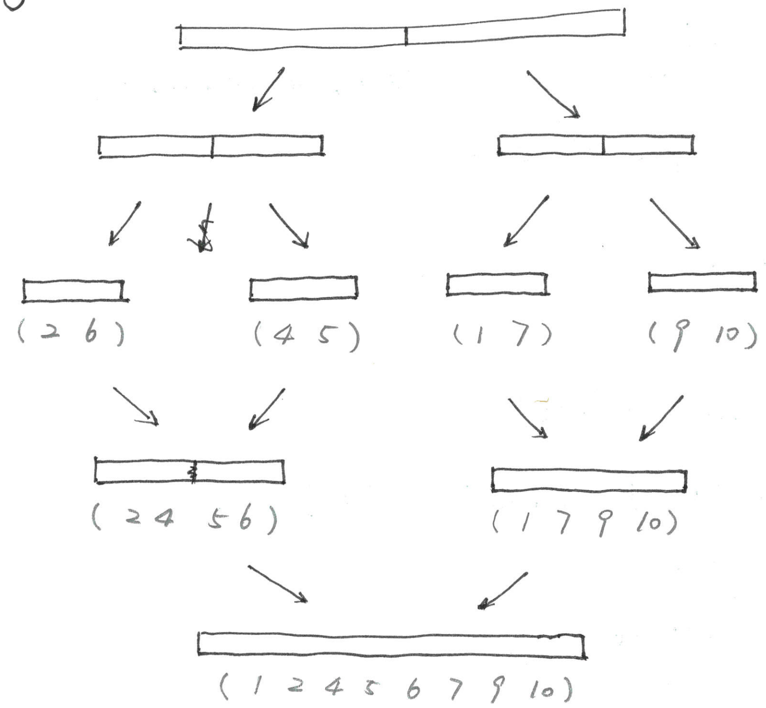


- Code :

```
// heapify
for ( int pos=1; pos<n; pos++ ){
    push_bst ( data, pos+1 , pos ); // ( int* head, size, to_be_inserted)
}
// change to sorted array
for( int pos = n-1; pos>0; pos-- ){
    swap ( &data[pos] , &data[n-1-pos] );
    "pop"_bst ( data, pos );   // pop ( head, size)
}
```

2. Merge Sort

- idea: divide and conquer (recurse on smaller pieces of the problem, then comes the results of the recursion)
- algorithm:



- code:

```
void mergeSort ( int * data, int size ){
    // base case: use selectionSort when n < 16
    if ( size <= n )
        selectionSort ( data, size );
    else {
        int left = size / 2 , right = size - size / 2 ;
        mergeSort ( data, left );
        mergeSort ( data + left, right );
        int tmp[ size ] ; // or int * tmp = new int[size];
        int pos = 0, pos_left = 0, pos_right = left ;
```

```
    while( pos_left < left && pos_right < size ){
        if( data[pos_left] < data[pos_right]){
            tmp[pos] = data[pos_left];
            pos_left ++;
        }
        else{
            tmp[pos] = data[pos_right];
            pos_right ++;
        }
        pos ++;
    }
    while ( pos_left < left){
        tmp[pos] = data[pos_left];
        pos++ ;
        pos_left ++;
    }
    while ( pos_right< right){
        tmp[pos] = data [pos_right];
        pos ++;
        pos_right ++;
    }
    for ( int i = 0; i< size ; i++ ){
        data[i] = tmp[i];
    }
    delete[] tmp;
}
```

3. Quick Sort

- idea : divide and conquer ;
  put "pivot" at its correct place (its lefts are all smaller and its rights are all larger) in each recursion.
- Algorithm :

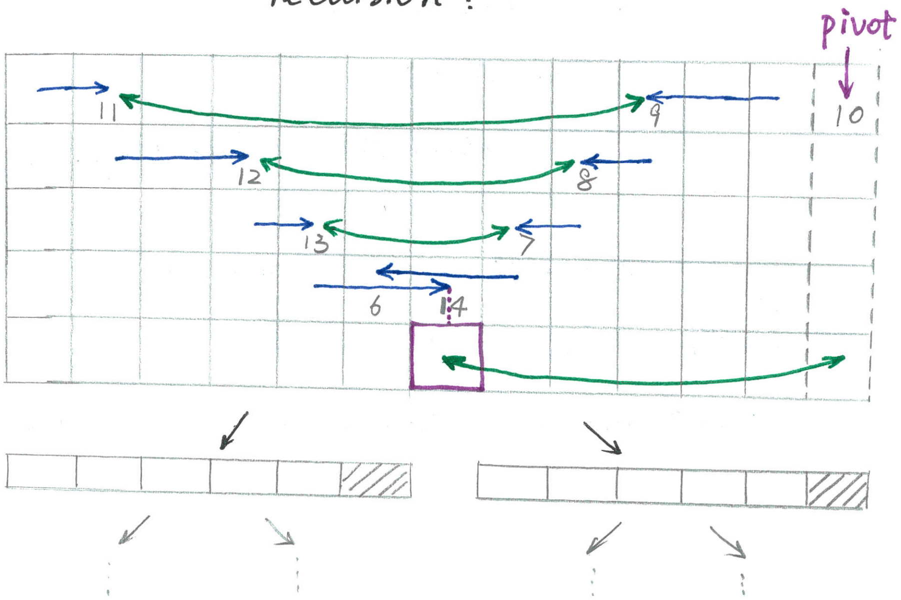
pick up a pivot with random array index

↓

and swap it with the last element of the array.

↓

recursion :



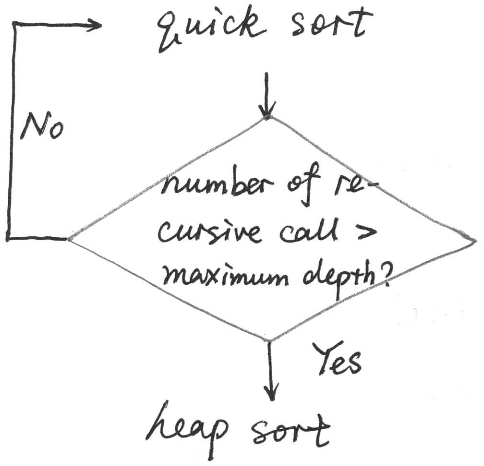
So if the array is nearly sorted, the complexity → $O(N^2)$, because pivot won't evenly split the problem.

- Code

```
void quickSort ( int * data , int size ){
        int pivotIdx = random ( size );
        swap ( & data[ pivotIdx ] , & data[ size-1] );
        quickSort_helper ( data , size ) ;
}
```

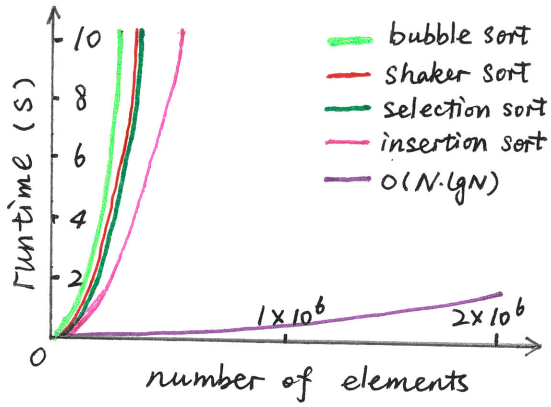
```
void quickSort_helper ( int * data , int size ) {
        // base case : e.g., n = 16 .
        if ( size <= n )
                selectionSort ( data , size );
        else {
                int pivot = data [ size-1 ] ;
                int left = 0 , right = size-2 ;
                while ( left < right ) {
                        while ( data[ left ] <= pivot )
                                left ++ ;
                        while ( data [right] >= pivot && right > left )
                                right -- ;
                        swap ( & data[left] , & data[right] ) ;
                }
                swap ( & data[left] , & data [ size-1 ] ) ;
                quickSort_helper ( data , left ) ;
                quickSort_helper ( data + left + 1 , size-left-1 ) ;
        }
}
```

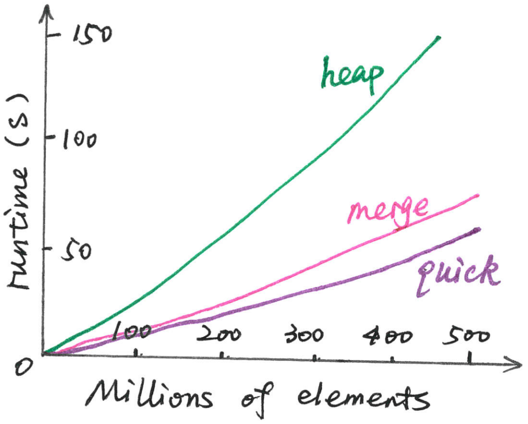
4. Introspetion Sort



Such hybridization guarantees O(NlgN) runtime, but still gives the performance benefits of quick sort in the general case.

■ Sorting efficiency & Trade-offs





	Heap Sort	Merge Sort	Quick Sort
Guaranteed $O(N \cdot \lg N)$?	Yes	Yes	No ($O(N^2)$ if well-order)
Extra Space Needed ?	$O(1)$	$O(N)$ (tmp array when merge two arrays)	$O(\lg N)$ (for stack frames)
Good locality ?	No (→ higher constant factors. due to BST bubbling up/down)	Yes	Yes

locality : ~~whether~~ accessing elements that are physically close to each other

■ Sorting Libraries

- in C :

```
void qsort ( void * base, size_t nmemb, size_t size,
                    int (* compar) ( const void* , const void* ) ) ;
// quick sort
```

- in C++ : ① std :: sort

  ② std :: stable_sort (equal inputs will remain the ~~ori~~ same order)

Ref : https://xkcd.com/1185/.

Chapter 27 Balanced BSTs

↳ maintain O(lgN) behavior

AVL Insertion

- AVL tree : ① store the height in a field in each node

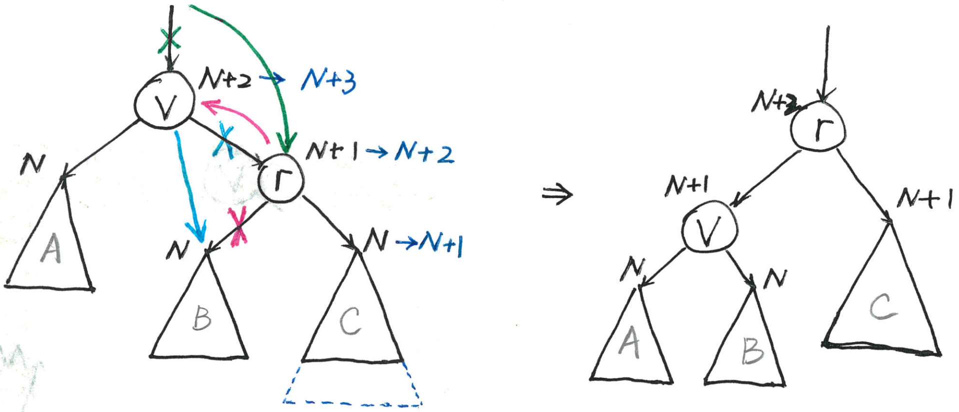
  ② perform rotation to restore the balance

  gurantee balance ; better for recursion.

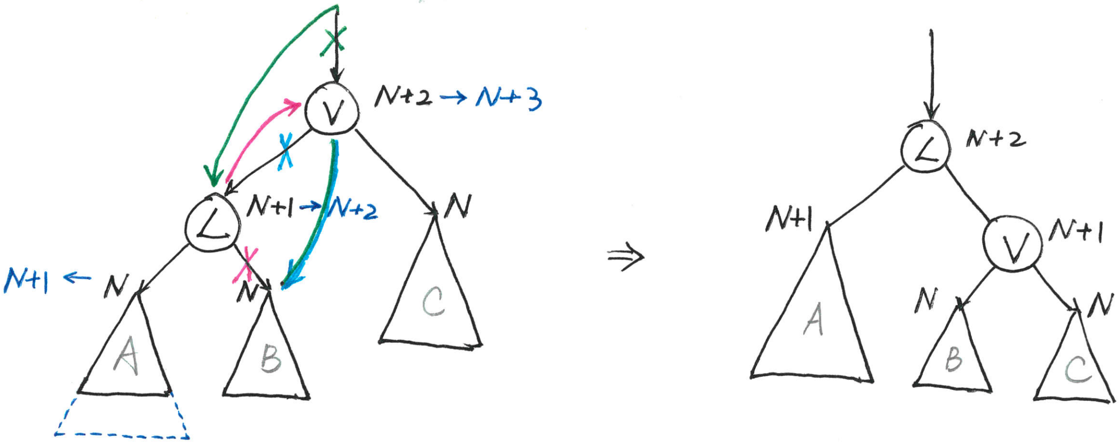
CASE 1 : Imbalance either arises from the right child's right child growing, or from left child's left child growing.

⇒ single rotations

- right's right → left rotation.



- left's left → right rotation

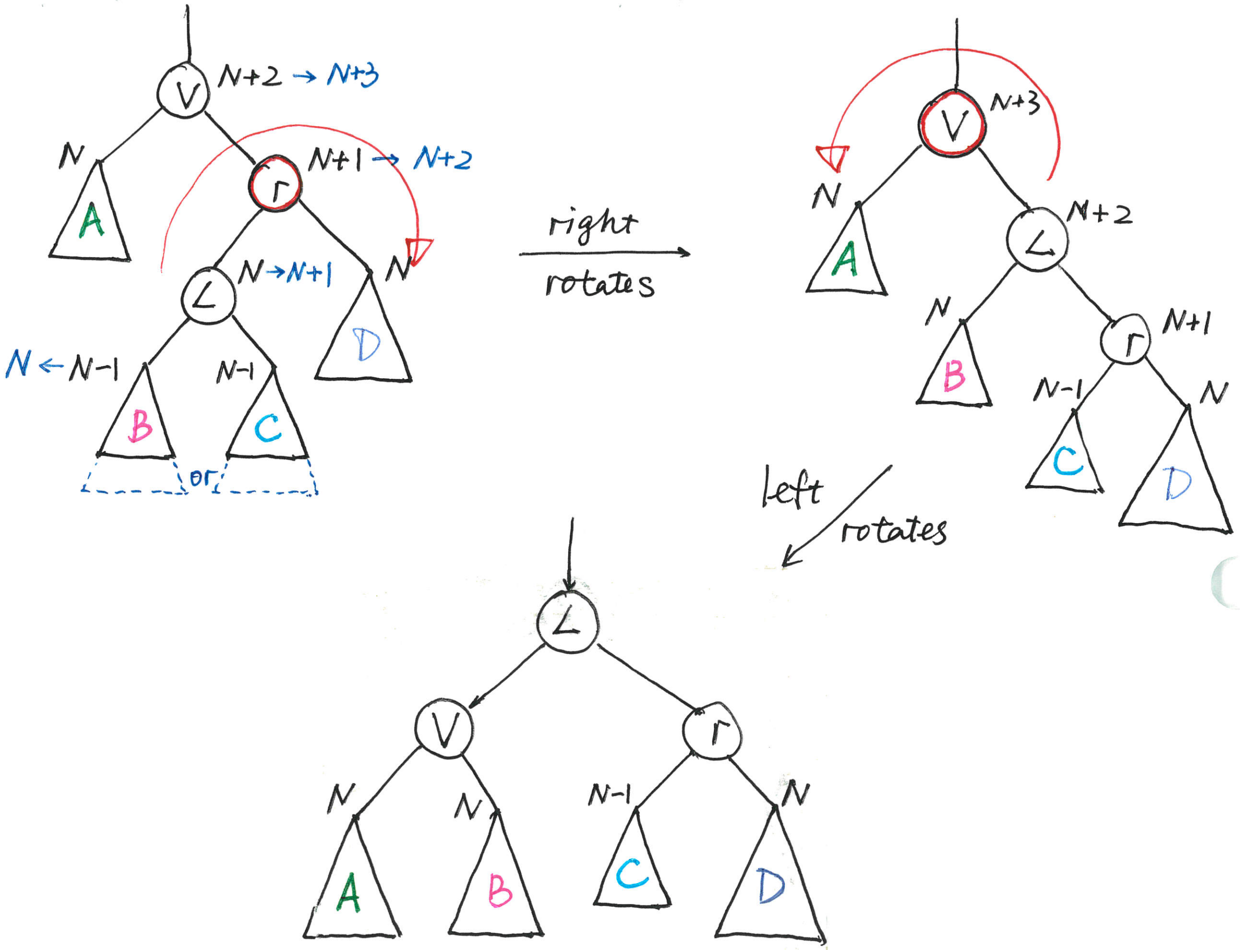


CASE 2 : Imbalance either arises from right child's left child,

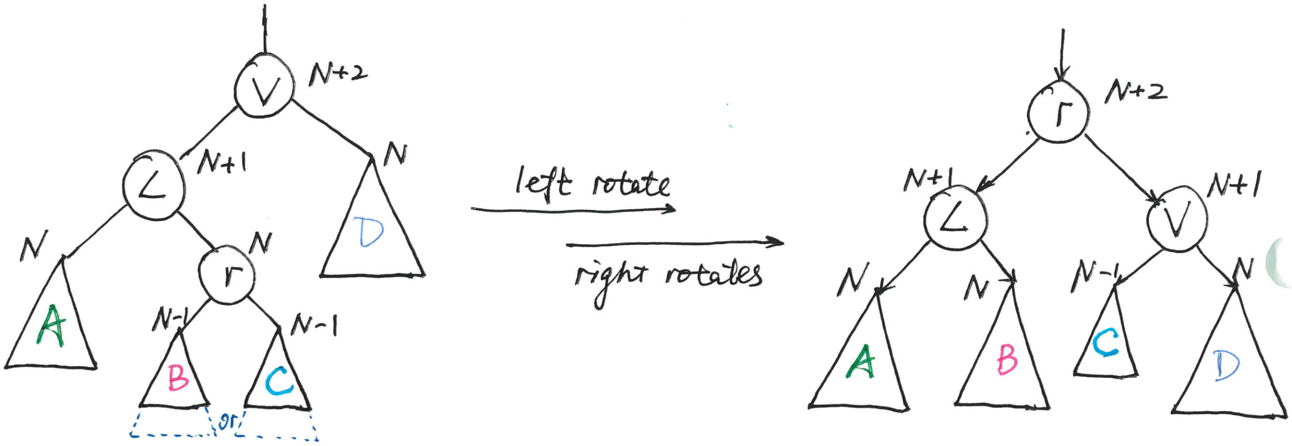
or from its left child's right child.

⇒ double rotations

- right's left : right rotates + left rotates.



- left's right : left rotates + right rotates.



Tentative Code 1 (not preferred) :

```
Node * addNode ( Node * curr , const K & key , const V & value , stack * path ) {

    if ( curr == NULL )
        return new Node ( key, value ) ; // height initialized as 0
    else {
        if ( key < curr-> key ) {
            curr-> left = addNode ( curr-> left , key, value, path ) ;
            // update height
            curr-> height ++ ;
            // ckeck for imbalance
            if ( curr-> left-> height > curr-> right-> height+1 ) {
                if ( path->top() == LEFT )
                    LsingleRotate ( & curr) ; // right rotation
                else
                    LdoubleRotate ( & curr) ; // left + right rotation
            }
            // update path ( a stack)
            path-> push ( LEFT ) ;
        }
       else if ( key > curr-> key) {
            Curr-> right = addNode ( curr-> right , key, value, path ) ;
            curr-> height ++ ;
            if ( curr-> right-> height > curr-> left -> height +1 ) {
                if ( path -> top () == RIGHT )
                    Rsingle Rotata ( & curr) ; // left rotation
                else
                    Rdouble Rotate ( & curr) ; // right + left rotation
            }
            path-> push ( RIGHT) ;
       }
       else
            curr-> value = value ;
  } return curr ; }
```

Tentative Code 2 (preferred) :

```
Node * addNode ( Node * curr, const K & key, const V & value ){

    if ( curr == NULL )
        return new Node ( key, value ) ;
    else {
        if ( key < curr-> key ){
            curr-> left = addNode ( curr->left , key, value );
            curr-> height ~~++~~ = max ( curr->left->height , curr->right->height ) + 1 ;
            if ( curr-> left->height > curr-> right-> height + 1 ){
                rebalance (& curr);
            }
        }
        else if ( key > curr-> key ){
            curr-> right = addNode ( curr-> right , key, value );
            curr-> height ~~++~~ = max (      ) + 1 ;
            if ( curr-> right -> height > curr-> left-> height + 1 ){
                rebalance ( & curr );
            }
        }
        else
            curr-> value = value ;
    }
    return curr ;
}
void rebalance ( Node ** V ) {
    if ( (*V) -> left-> height > (*V) -> right -> height ) {
        // left's left
        if ( (*V)->left-> left ->height > (*V)-> left-> right-> height )
            rightRotat~~ion~~ ( V ) ;   // take care of modifying weight
        // left's right
        else{
            leftRotate ( & ( (*V)-> left )) ;
            rightRotate ( V );
        }
    }
```

Code:

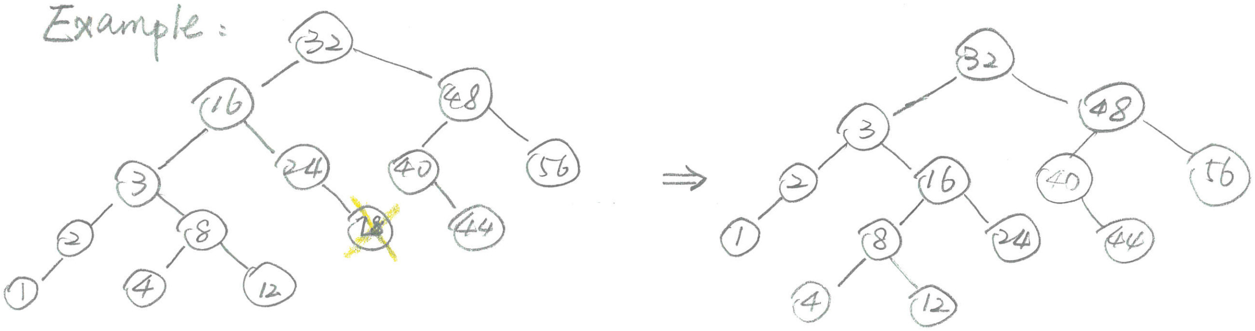
```
Node * addNode ( Node* curr, const K& key ){
    if ( curr == NULL )
        return new Node ( key );
    if ( key < curr->key ){
        curr->left = addNode ( curr->left, key );
        if ( getHeight ( curr->left ) > getHeight ( curr->right ) + 1 ){
            // left's left
            if ( getHeight ( curr->left->left ) >= getHeight ( curr->left->right ) ){
                rightRotate ( & curr );
            }
            // left's right
            else {
                LeftRotate ( & curr->left );
                rightRotate ( & curr );
            }
        }
        updateHeight ( curr );
    }
    else if ( key > curr->key ){
        curr->right = addNode ( curr->right, key );
        if ( getHeight ( curr->right ) > getHeight ( curr->left ) + 1 ){
            // right's right
            if ( getHeight ( curr->right->right ) >= getHeight ( curr->right->left ) ){
                LeftRotate ( & curr );
            }
            // right's left
            else {
                rightRotate ( & curr->right );
                leftRotate ( & curr );
            }
        }
```

```
            updateHeight( curr);
        }
        return curr;
    }
```

AVL Deletion

idea: after actually deleting a node, check the imbalance of it's parent and grandparents, and update height.

Example:



Code:

```
Node * removeNode ( Node * curr, const K & key) {
    if ( curr == NULL)
       return curr;
    if ( key < curr-> key) {
       curr-> left = removeNode ( curr-> left, key);
       if ( getHeight ( curr-> right) > getHeight ( curr-> left) + 1) {
           // right's right
           if( getHeight ( curr-> right-> right) >= getHeight ( curr-> right-> left ) ) {
               leftRotate ( & curr);
           }
           ~~ed~~ // right's left
           else {
               rightRotate ( & curr-> right);
               leftRotate ( & curr);
           }
       }
```

```
            updateHeight( curr);
        }
        else if ( key > curr->key ) {
            curr-> right = removeNode ( curr->right, key);
            if ( getHeight( curr-> left) > getHeight ( curr->right )+1 ){
                // left's left
                if (getHeight( curr->left->left) >= getHeight( curr->left->right ) ){
                    rightRotate (& curr) ;
                }
                // left's right
                else {
                    leftRotate (& curr->left) ;
                    rightRotate ( & curr) ;
                }
            }
            updateHeight ( curr);
        }
        else {
            if ( curr->left == NULL ) {
                Node * tmp = curr->right ; delete curr ; return tmp ;
            }
            else if ( curr-> right == NULL ) {
                Node * tmp = curr -> left ; delete curr ; return tmp;
            }
            else {
                Node * tmp = curr->left ;
                while ( ~~curr~~ tmp -> right != NULL )
                        tmp = tmp-> right ;
                Curr-> key = tmp-> key ;
                curr -> left = removeNode ( curr-> left, tmp-> key ) ;
            }
        }
        return curr;
    }
```

```
    else {
        // right's right
        if ( (*V)->right->right->height > (*V)->right->left->height )
            leftRotate (V);
        // right's left
        else {
            rightRotate ( &( (*V)->right) );
            leftRotate (V);
        }
    }
}
```

AVL Deletion

idea: after deleting a node (the really deleted one, not necessarily the one that holds the target value), check the imbalance of its parent and grandparents. Update height.

Tentative Code:

```
void remove ( const K & key ) {
    Node ** curr = & root;
    ~~Node ** prev = NULL;~~  Stack * path = new stack();
    while ( * curr != NULL && (*curr)->key != key ) {
        ~~if (key < (*curr)->key)~~
        ~~prev = *curr;~~   path-> push (*curr);
        if ( key < (*curr)->key )      curr = &( *curr)-> left;
        else                           curr = & ( * curr)-> right;
    }
```

```
if ( * curr != NULL ) {
    if (  (*curr) -> left  == NULL ) {
        Node * tmp  =  * curr ;
        * curr  =  tmp -> right ;
        delete tmp ;
        ~~if ( prev != NULL && prev -> height > 1 )~~
    }
    else if (  (*curr) -> right == NULL  ) {
        Node * tmp  =  * curr ;
        * curr  =  tmp -> left ;
        delete tmp  ;
    }
    else {
        Node ** toRp =  & ( * curr) -> left ;     path -> push ( * curr) ;
        while ( ( * toRp ) -> right != NULL ) {
            path -> push ( * toRp ) ;
            toRp  =  ( * toRp ) -> right ;
        }
        ( * curr ) -> key  =  ( * toRp ) -> key ;
        ( * curr ) -> value  =  ( * toRp) -> value ;
        Node * tmp =  * toRp ;
        * toRp  =  tmp -> left ;
        delete tmp ;
    }
    while (  ! path -> empty() ) {
        ~~if(~~check Balance ( &path -> top() ) ~~}~~ ;
        path -> pop() ;
    }
}

}
```

```
void checkBalance ( Node **V) {
    if ( (*V)->left == NULL) {
        if ( (*V) -> right == NULL )
            (*V) -> height = 0 ;
        else {
            if ( (*V) -> right-> height > 1) {
                leftRotate (V) ;
            }
        }
    }
    else {
        if ( (*V) -> right == NULL) {
            if ( (*V) -> left-> height > 1 )
                rightRotate (V) ;
        }
        else {
            if ( (*V) -> left-> height - (*V) -> right-> height > 1
                || (*V) -> right-> height - (*V) -> left-> height > 1 )
                rebalance (V) ;
          (*V) -> height = max ( (*V) -> right-> height , (*V) -> left-> height ) + 1 ;
        }
    }
}
```

Red / Black Insertion

- Red / black tree : ① do not guarantee balance, but ensure the maximum length of a path from the root to a leaf is O(lg N).

  ② amenable to iterative implementations

- Invariants : ① every node is either red or black

  ② root is black (usually consider NULL as black as well)

  ③ if a node is red, its children must be black.

  ④ every root → NULL path must have the same number of black nodes.

⇒ worst case : shortest~~case~~ = n black , longest path = n black + n red.

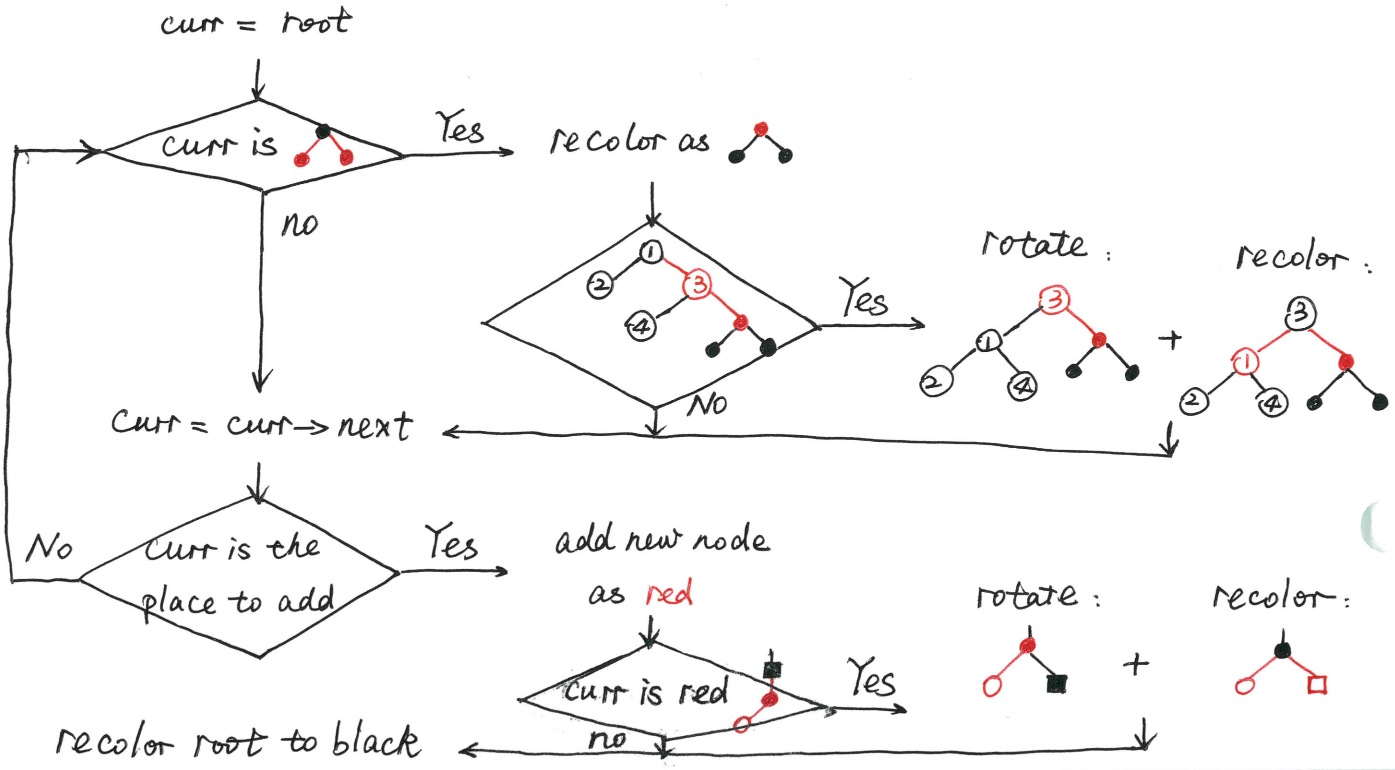
② ⇒ recolor root to black after every insertion.

③ ⇒ rotate + recolor if two red nodes connect:

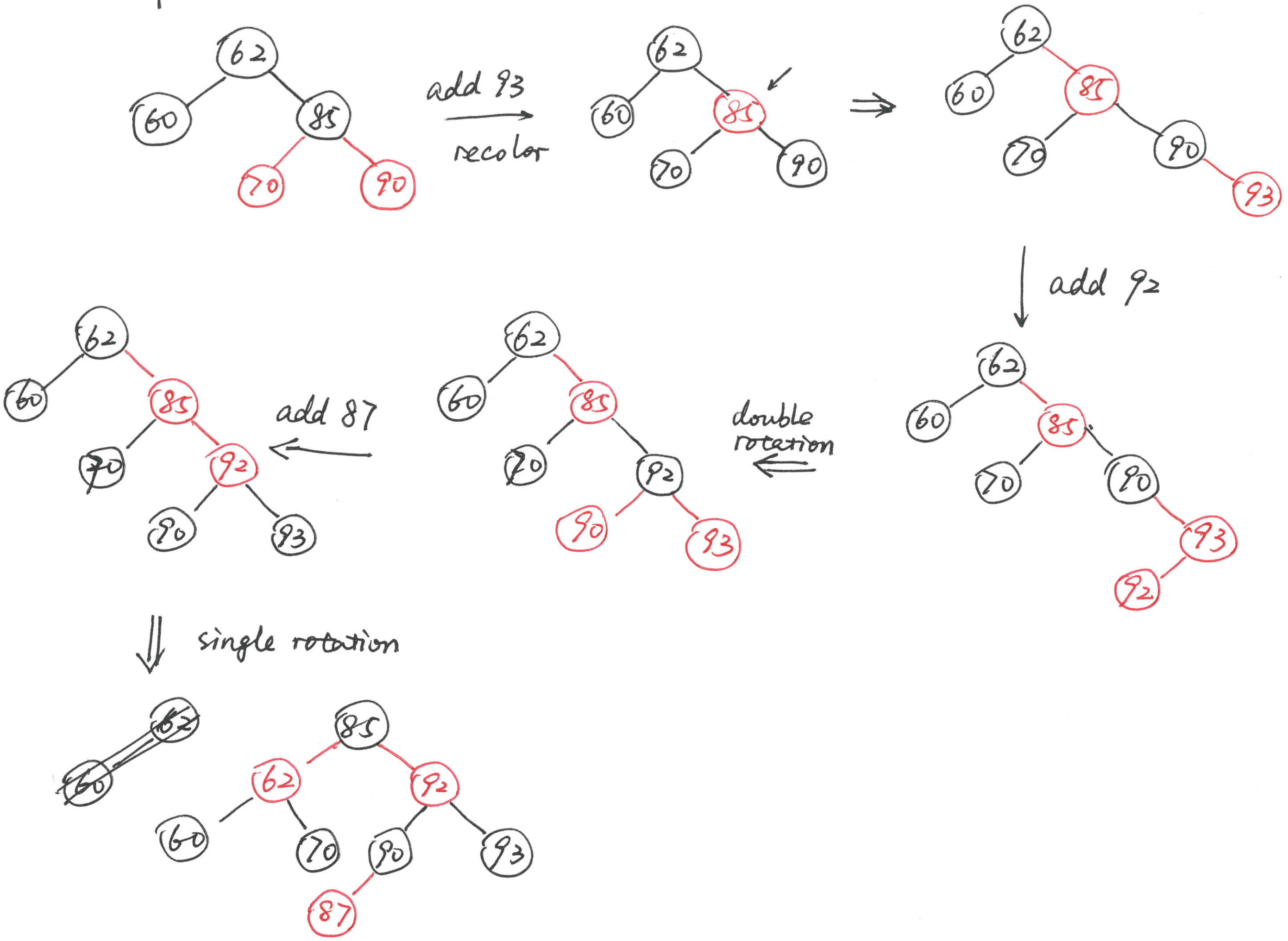
④ ⇒ The only way to increase the black-count of paths is to recolor root as black;

Each recolor is 

- Algorithm :



Example:



■ Red / Black Deletion

- Red node: simply delete it.
- Black node: Matt Might's approach

new color scheme:

−1	0	1	2
(negative black)	(red)	(black)	(double black)

① set the deleted NULL as 2.

② push the "double black" node up the tree towards to root, and

— increase the blackness of its parent by 1

— decrease - - - - - - - - itself and its sibling by 1

③ rotate / ~~eliminate via~~ recolor to eliminate violations

④ set root to black.

Chapter 28 Concurrency

■ Run multiple **Process** concurrently

process : a running instance of a program.

Feature :
- completely independent to each other (code is the same)
- each has its own execution arrow and memory space
- identified by its PID (process ID) . when you want to make system calls that manipulate the processes.

1. use a combination of fork, execve and waitpid

   ↳ to run other program in parallel, e.g., used in shell.

   pid-t fork ( );

   - create a new process (child process) which is exactly like the original process (parent process) , except for :

     ① the return value = { child : 0 ; parent : child's PID

     and -1 if encounters an error.

     ② their PIDs

     ③ their parent 's PID

     ④ resetting some OS- level resource accounting.

   - Memory and file descriptors are copied. (two independent sets)

   execve ~~execute~~ (char* program. char ** argv. char ** envp) ;

   - replace the program in the currently running process with a newly loaded program (1st argument)
   - do not return on success
   - If called by a child process, the second copy of the parent's memory is destroyed. and replaced with a memory image loaded from the binary.

pid_t ~~int~~ waitpid (pid_t cpid, int * status, int options)

- allow a parent process to wait for one or more of its child process to terminate.

Example:

```
void forkProcess ( char* program, char** argv, char ** envp ){
    pid_t cpid, w;
    int status;

    cpid = fork();
    if ( cpid == -1 ) { perror("fork"); return;}
    if ( cpid == 0) {        // code that will be executed by parent.
        execute ( program, argv, envp );
        perror( program ) ;   // execve only returns an error
        exit ( EXIT_ FAILURE ) ;
    }
    else {
        do {
            w= waitpid ( cpid, &status, WUNTRACED | WCONTINUED );
            if ( w==-1) { perror("waitpid") ; return;}
            // may check the ~~exe~~ child's return value.
                :
        } while ( ! WIFEXITED (status) && !WIFSIGNALED (status) );

        // Do some other things
    }
}
```

2. Use fork() alone to complete tasks in parallel. e.g., webserver

Example:

```
if ( cpid== 0) {
    handleRequest (r);
    exit ( EXIT_ SUCCESS) ;
}
else { // do whatever else the parent need to do. }
```

■ Threads

thread : a single sequence stream within a process.

Feature :
- share an address space (heap, code segment, global data segment)
- each has its own stacks and execution arrow.
- one thread can access data in the other's stack via pointers.

1. Creation

`# include < pthread.h >` ; `-lpthread`.

```
int pthread_create ( ¹pthread_t * thread,
                     ²const pthread_attr_t * attr,
                     ³void * ( * start_routine) ( void * ), // function pointe
                     ⁴void * arg );
```

(1) Arguments :

① filled in by pthread_create so that it can be used later for identification

② attributes of the thread. NULL for default.

③ Entry - point (entry function) for the new thread to start executing. The new thread exits whenever it returns.

④ arguments. that to be passed into the entry function.

(2) Things happened during creation :

① A new stack is created, which is independent from the caller's, with a frame for the entry function, and passing in its args.

② A second execution arrow is created at the start of the entry function, by making a system call to spawn a new thread.

2. Execution in Parallel

* multi-threaded programs are non-deterministic :
there is a set of behaviors rather than one particular behavior
which we will observe for a specific input.

3. Thread Exit (stop running)

(1) A thread exits when : entry function returns,

or : parent calls `void pthread_exit(void* retval)`

(2) How to deal with a created thread :

① Wait for it to exit and obtain its return value

↓ "join"

↓ either what the entry function returns
or the argument passed to pthread_exit.

by calling `int pthread_join(pthread_t thread, void ** retval)`

↓ can be NULL if not care return value.

When it is called, the thread blocks (its execution arrow stops advancing) until the thread to be joining terminates. And the pthread library will release its resources.

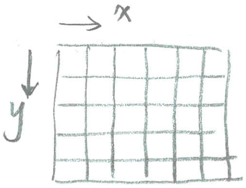
② tell the pthread library that it will never join another thread by calling `int pthread_detach(pthread_t thread)`.

When it is called, the library will release the resources as soon as the thread exits.

Note : if main() returns, the entire process exits, terminating all of the thread inside of it.

Example : smooth an image, from * src to * dst.

- Sequential :



```
void smooth( image_t * src, image_t * dst) {
    for ( int y=0 ; y< src-> height ; y++ ){
        for( int x= 0 ; x< src-> width ; x++ ){
            dst-> data [y][x] = wavg_pixel (src,x,y);
        }
    }
}
```

- Parallel :

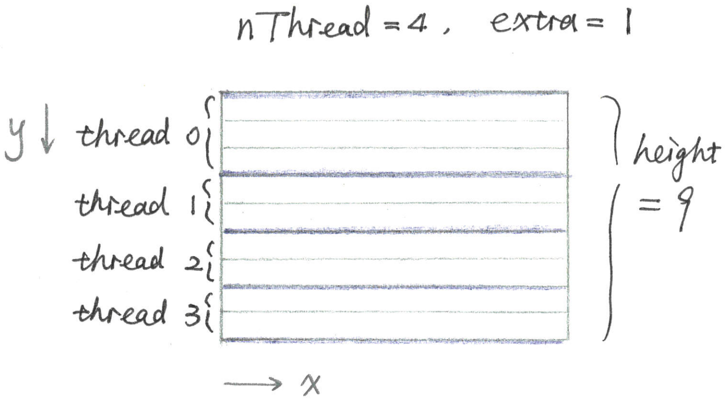
```
// conbine all info into a single variable, as requested by pthread_create
typedef struct {
        image_t * src ; image_t * dst;
        int startY ; // defines the tasks for each thread.
        int endY ;
} thr_arg ;

// entry function

void * smoothThread ( void* varg ){

    thr_arg * arg = varg;

    for( int y = arg-> startY ; y< arg-> endY , y++ ){

        for ( int x= ~~ar~~ 0 ; x < src-> width ; x++ ){

            arg-> dst-> data [y][x] = wavg_pixel ( arg-> src, x.y);
        }
    }
    free ( arg) ; // Don't do that before it finishes by worker
    return NULL ;
}
```

---

4. Choice of nThreads

① No more that what the hardware supports.

② Fewer threads for smaller problem size, to avoid overhead.



```
void smoothParallel ( image_t * src,
                      image_t * dst,
                      int nThreads ) {

  int perThread = src->height / nThreads
                  +1 ;
  int extras = src -> height % nThreads ;
  int curr = 0 ;
  pthread_t * threads = malloc ( nThreads * sizeof ( * threads) ) ;
  // spawn tasks to each thread.
  for ( int i=0 ; i< nThreads ; i++ ) {
       if ( i== extras )   perThread -- ;
       thr_arg * arg = malloc ( sizeof ( * arg ) ) ;
       arg -> src = src ;   arg -> dst = dst ;
       // define boundary for each thread.
       arg -> startY = curr ;
       arg -> endY  =  curr + perThread ;
       curr += perThread ;
       pthread_create ( & threads [i] , NULL, smoothThread , arg ) ;
  }
  for ( int i= 0 ; i< nThreads ; i++ ) {
       pthread_join ( threads [i] , NULL ) ;
  }
  free ( threads ) ;
}
```

■ Synchronization

- parallel problem can be
  - **embarrassingly parallel (EP)** : there are apparent ways to form completely & independent jobs.
  - use some **synchronization** : a technique in which threads are forced to wait before performing specific operations.

- Synchronization is require when there is **critical section** exist, i.e., a region of the program that we must ensure at most one thread's execution arrow is inside of at any given time.

- Otherwise the program will have **data races** — situations in which multiple threads are accessing the same data, and the specific order in which we advance the execution arrows of each thread affects the results.

Two useful synchronization constructs : mutexes & barrier.

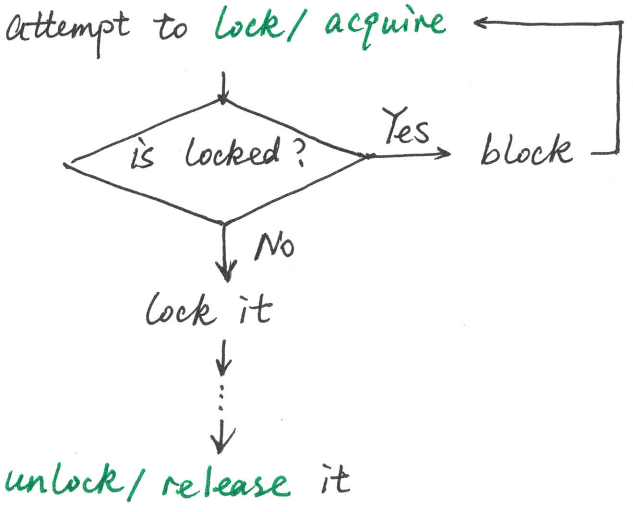
1. Mutex (Lock) ;

(1) **mutual exclusion lock / mutex / lock** : guard a critical section.

two states { locked / unlocked (initialized state) } for a particular piece of data.

" lock a piece of data " ≡ lock the mutex that guards that piece of data.

" hold a lock on a piece of data " ≡ ~~e~~ put a mutex which guards that data.



Example :

```
pthread_mutex_t  lock ;

void * incrThread ( void *varg ) {
    int * arg = varg ;
    for ( int i=0 ; i< 5000 ; i++) {
        pthread_mutex_lock ( & lock ) ;
        int tmp = * arg ;
        tmp ++ ;
        * arg = tmp ;
        pthread_mutex_unlock ( & lock ) ;
    }
    return NULL ;
}
```

```
int main() {
    int x= 0 ;
    pthread_t thr ;
    pthread_mutex_init (& lock, NULL) ;
    pthread_create ( & thr, NULL, incrThread, & x);
    incrThread ( & x) ;
    pthread_join ( thr, NULL);
    pthread_mutex_destroy (& lock);
    printf ( "%d\n", x);    // ans = 10000
    return EXIT_SUCCESS ;
}
```

- performance issue :

  ① locking a mutex has a non-trivial overhead (as a result of cache coherence). so it's not good for a largely sequential code .

  ② Heavily contended mutex will significantly serialize the code .
  ↳ (many threads are trying to acquire it at once)

- Improvement : change our locking granularity (how large of a piece of data protected with a single lock), by choosing a more suitable data structure .

  e.g., a hashtable with one mutex per bucket . Threads will only be ~~speci~~ serialized if the access the same bucket , which is hopefully rare .

  or include a function in map's interface which takes a function pointer of " what to do to the value " if it's locked .

Trade-off : finer-grained locking leads to better scalability , but requires more cares from the programmer .

(2) **Reader / Writer Locks** : only writing the guarded data needs exclusive access.

↳ { reading : allow multiple threads to simutaneously lock it.
  | writing : ① only one thread can lock it ② may not do so when any other thread holds a read lock.

Great if common operations needs to only read the data.

Example:

```
pthread-rwlock-t  bucketLock;
pthread_mutex_t  locks [NUM_OF_BUCKET] ;
void add ( K & key, V & value){
    int ind = hash(key);
    pthread_rwlock_rdlock ( &bucketLock);
    ind = ind % numBuckets;
    pthread_mutex_lock ( &locks [ind]);
    buckets [ind] -> add ( key, value);
    pthread_mutex_unlock ( & locks[ind] );
    pthread_rwlock_unlock ( & bucketLock);
}
```

(3) **Conditional Variable** : supports operation of wait/signal/broadcast

↳ have a thread to wait for a particular condition to become true before it proceed.

~~wait~~ ① `int pthread_cond_wait ( pthread_cond_t * restrict cond, pthread_mutex_t * restrict mutex);`

blocks the thread until some other thredd does a signal / broadcast.

② `int pthread_cond_signal ( pthread_cond_t * cond );`

unblocks ONE thread that is waiting.

③ `int pthread_cond_broadcast ( pthread_cond_t * cond);`

unblocks ALL thread that is waiting.

While the thread starts to wait, another thread may aquire the mutex before the thread reaquires it.

Syntax:

```
WorkItem * getWork ( Queue * myQueue ){
    pthread_mutex_lock ( & myQueue -> lock );
    while ( myQueue -> isEmpty() ){
        pthread_cond_wait ( & myQueue -> cv, & myQueue -> lock );
    }
    WorkItem * answer = myQueue.dequeue();
    pthread_mutex_unlock ( & myQueue -> lock );
}
void addWork ( Queue * myQueue, WorkItem * w){
    pthread_mutex_lock ( & myQueue -> lock );
    myQueue -> enqueue (w);
    pthread_cond_signal ( & myQueue -> cv );
    pthread_mutex_unlock ( & myQueue -> lock );
}
```

(4) Things that should be avoided:

① **Dead lock**: one or more threads cannot advance at all because they are waiting for a locked mutex which will never be unlocked.

Example:

```
Thread 0:                      Thread 1:
  lock ( & lockA );              Lock ( & LockB );
→ lock ( & lockB );            → Lock ( & LockA );
      ⋮                              ⋮
  unlock ( & lockA );            unlock ( & lockB );
  unlock ( & lockB );            unlock ( & lockA );
```

To avoid deadlock:

(i) always acquire locks in the same order

(ii) use **pthread_mutex_trylock**, which does not block if mutex is lock, but instead returns a non-zero value.

(iii) cannot busy wait while we hold a mutex if some other thread must aquire that mutex to satisfy the condition we are busy waiting on.

② **Busy wait**: executing a loop that does nothing until some condition is met.

Example:

```
WorkItem * getWork( Queue * myQueue) {
    pthread_mutex_lock ( & myQueue-> lock );
    while ( myQueue -> isEmpty ()) {
        pthread_mutex_unlock ( & myQueue -> lock );
        // do something that takes some time ?
        pthread_mutex_lock ( & myQueue-> lock);
    }
      ⋮
}
```

2. Barriers

barrier: a construct which requires a certain number of threads to reach it before any may proceed.

Threads that first arrive at the barrier will block until the required number of threads reaches the barrier.

Significantly useful in scientific computing.

■ Atomic Primitives

↳ operations that cannot broken up into smaller parts that could be performed by different process, and are guaranteed isolated.

1. Test-and-Set (TAS)

↳ atomatically test (read) the value in a memory location and set it to 1.

```
≈ int test_and_set (int *ptr) {
        int x = *ptr ;   // test
        *ptr = 1 ;       // set
        return x ;
  }
```

one concern : the hardware is generally allowed to reorder memory operations (read & write) to improve performance within " memory consistency model ".

Improvement : test-and-test-and-set

↓

test the lock with a non-atomic operation. then only use TAS when there's possibility that we might actually acquire the lock.

Motivation : TAS requires the processor to obtain exclusive access permission to that data, which is a bit expensive.

Example :

```
typedef int mutex_t ;
void mutex_lock ( mutex_t * lock ) {
    do {
        while ( * lock != 0 ) {  // test
            asm volatile ("pause\n"); // ask p to wait a bit via a special instruction
        }
    } ~~while~~ while( test_and_set ( lock ) != 0 );  // TAS
}
```

2. Compare-and-swap (CAS)

↳ more genaral :

```
original_value = *location;
if (original_value == expected_value){ //compare
    *location = new_value;  // swap
}
return original_value;
```

compare_and_swap (lock, 0, 1); ⟷ test_and_set (lock);

expected_val new_val

3. Load-linked & Store_conditional

: used to build operations which behaves atomically.

↓ reads a value from a memory & asks the hardware to watch that memory address

↘ writes to that watched memory only if it has not be changed. otherwise it fails.

Example : build TAS operation.

```
int test_and_set (int *ptr){
    int temp;
    do{
        temp = load_linked(ptr);
    } while ( store_conditional (ptr, 1) != SUCCESS );
    return temp;
```

4. Atomic Increment

↳ atomically reads a value from memory, adds 1, stores it back.

Generally much more efficient than acquiring a lock, performing the increment, and releasing the lock.

Lock Free Data Structures

↳ which are capable of operating correctly when multiple threads access them simultaneously, but which do not need locks to privide the correctness,

- Rely on the use of atomic primitives (e.g., CA~~P~~S)
- Work best when there are not many racing writes.
- possible problem :
  - ① some operations may internally implemented using lock. e.g., new.
  - ② delete ptr ; may lead to others' use of dangling pointers, and may exist other racing problems.

Parallel Programming Idioms.

- Data parallelism :
  - different data elements can be processed in an independent fashion.
  - one way : use vector instructions (which performs the same operation on multiple pieces of data at one time)
- Pipeline parallelism.
  - divide our program into a series of tasks that are carried out in an assembly-line-like fashion.
- Task parallelism.
  - task : an invocation of a function with a particular argument
  - tasks spawn more tasks, and wait for their child tasks' results when they need them.
  - for divide-and-conquer algorithms.

- things to consider :

  load balancing : make sure one threads does not sit idle while other's working
  ↓
  work stealing : a thread whose task queue runs empty will steal work from the head of another thread's task queue.

■ Amdahl's law

$$\text{speedup}(N) = \frac{S+P}{S+\frac{P}{N}}$$

where S – the time of the serial portion. P – the time for the paralleled portion
N – number of threads.

⇒ make the common case (what happens most of the time) fast.

Chapter 29 Advanced Topics in Inheritance

Object Layout

- **Sub-object rule** (for fields & vitual (vtable) method) :

  the way we lay out a child class must contain a complete ~~sb~~ subclass of its parent class.

- fields : placed in the memory one after another.
  in the same position relative to the start of the object.

  non-virtual field : inrelavant to the object layout.

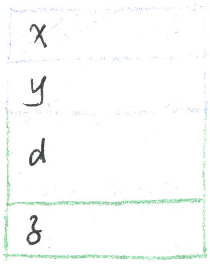
  virtual field : store a single pointer to **vtable** (virtual function table)

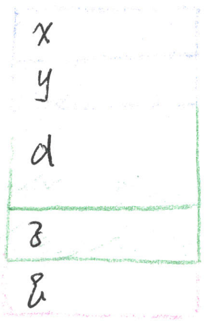
- vtable : a table of function pointers (one pointer per vitual function).

  All object of the same dynamic type will have their vtable pointers point at the ~~smo~~ same vtable.

  Vitual destructor is also an entry in vtable, and the same entry will be used for the destructor in all children.

Example 1 : inheritance :



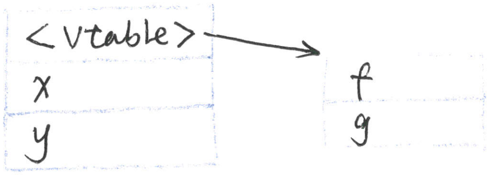


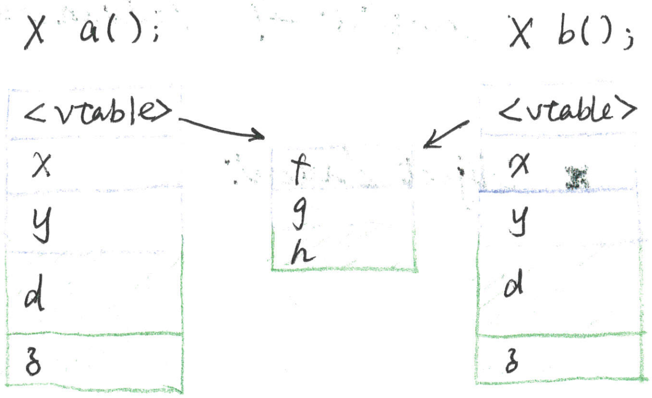
```
class A {
    int x;
    int y;
}
```

```
class B : public A {
    double d;
    int z;
}
```

```
class C : public B {
    int q;
}
```

Example 2 : with virtual method.





```
class X {
    int x;
    int y;
    virtual void f();
    virtual void g();
};
```

```
class Y : public x {
    double d;
    int z;
    virtual void f();  // overload
    virtual void h();  // new one
};
```

- Static v.s. dynamic dispatch.

static dispatch	dynamic dispatch
an instuction that calls a target which is known at compile time	1. reading memory to get the vtable pointer 2. a load from vtable to get the actual function pointer 3. an instruction which actually calls the function via that pointer.

Overall : The performance costs for them have relatively small difference.

But : ① dynamic dispatch prevents the compiler from inclinig call (an optimization that the compiler directly put short function instructions in the caller's function).

② the processor may have difficulty predicting the target of the functiona call in advance.

Multiple Inheritance

↳ a class that inherits from more than one parent class, and may treated polymorphically as any of its parents.

1. Syntax :

```
class ImageButton : public Button, public ImageDisplay {
                          ↓
          ⋮         primary parent
};
```

Note : inheriting from the same class twice is illegal.

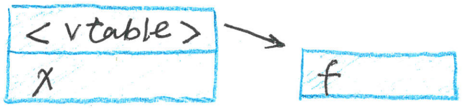
possible conflict : two parents have the same name for a field / method. In this case, use fully qualified name to eliminate ambiguous. Otherwise, use virtual inheritance instead if applicable.

2. Construction / Destruction.

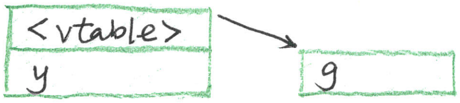
* Every parent's class's constructor is called in the order that the inheritance is declared. Not the one in the initializer list.

Then other rules for construction / destruction are similar to single inheritance

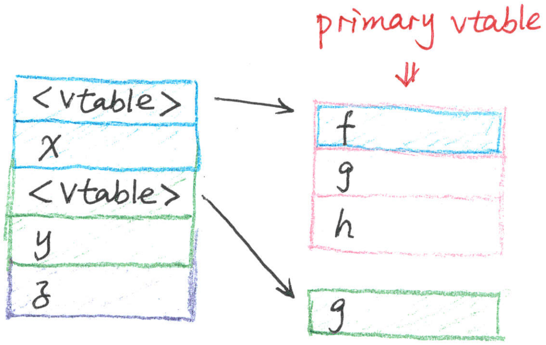
3. Layout



```
class A {
      int x;
      virtual void f();
}
```



```
class B {
      int y;
      virtual void g();
}
```



```
class C : public A, public B {
      int z;
      virtual void h();
}
```

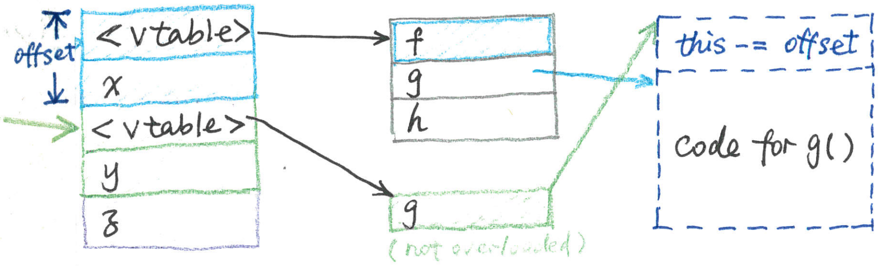
4. Conversion from primary vtable to non-primary parent's vtable.

① when we call a method inherited from a non-primary parent.

② during ~~ins~~ construction / destruction when that of a non-primary parent is invoked.

5. Overridden function via a non-primary parent.

```
void someFunc ( B* aB ){
    aB -> g();
}
void anotherFunc ( C* aC ){
    someFunc ( aC );
}
```



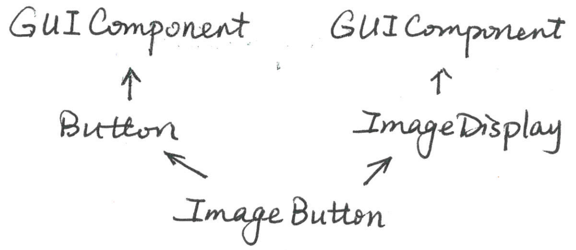
```
class C : public A, public B {
    int z;
public:
    virtual void h() {...}
    virtual void g() { // overridden from B
        h();
        z++;
    }
};
```

The compiler arranges for the non-primary vtable to point at a few instructions which precede the code of g(), and adjust "this" appropriately, such that the actual called g() is the same as in the primary vtable.

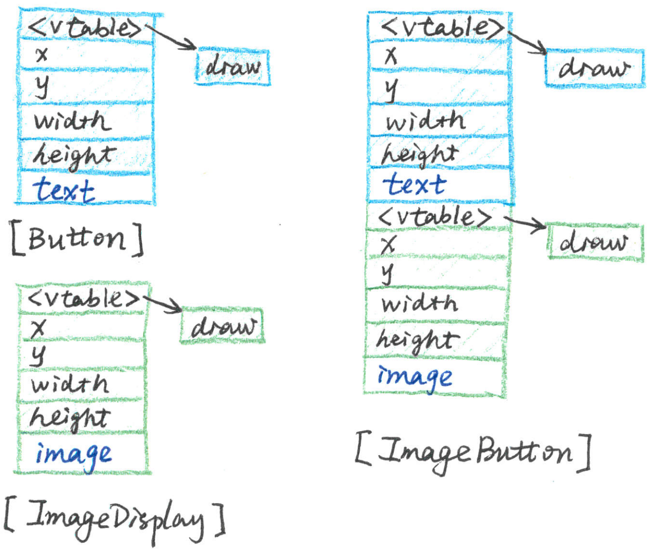
Virtual Multiple Inheritance

virtual inheritance : inherit from a parent class in such a way that only one sub-object of that parent class appears in any future descendants.

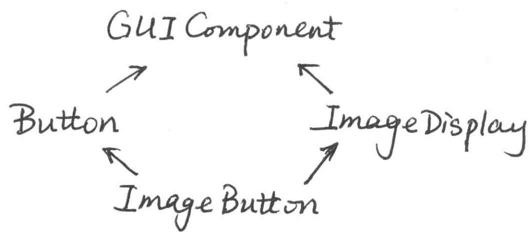
simple multiple inheritance



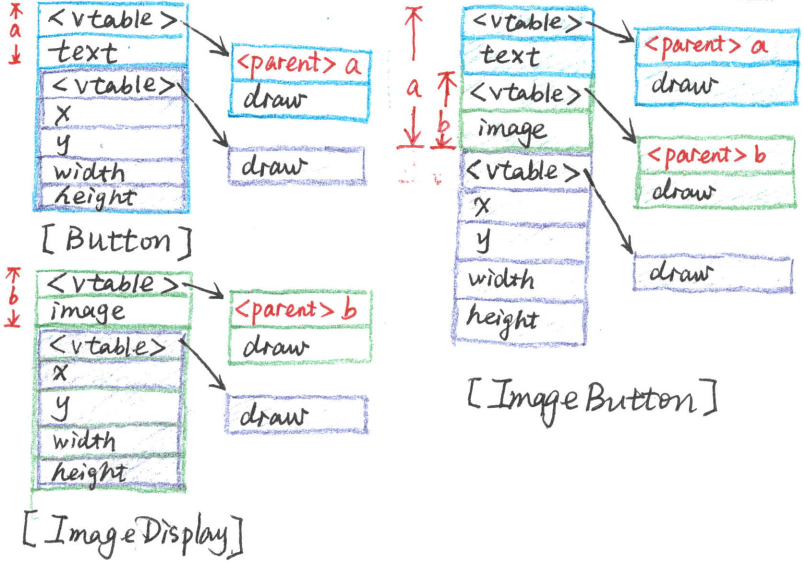
```
class GUI Component {
    protected :
        int x ; int y ; int width; int height;
    public :
        virtual void draw ( ) ;
};

class Button : public GUIComponent {
    string text ;
        ⋮
};
class ImageDisplay : public GUIComponent {
    protected :
        Image * image ;
            ⋮
};
```



virtual multiple inheritance



```
class Button : public virtual GUIComponent {
    string text;
        ⋮
};
class ImageDisplay : public virtual GUIComponent {
    protected :
        Image * image;
            ⋮
};
```



- Construction / Destruction:

  1. virtually inheritted classes' constructors are invoked first, before non-virtually inherited ones.
  2. Draw the hierarchy DAG.
  3. Depth-first-searth starting from the newly created class, record the post-order number of each class.

■ Mixins

↳ write a sub-class which can extended from a variety of super classes. so that the common functionality defined in that sub-class is at the bottom of inheritance hierarchy.

Example:

```
template < typename Parent >
class FancyBorderedComponent : public Parent {
    public :
      virtual void draw( ){
          Parent :: draw ( );
             ⋮
      }
};
```