Cheat Sheet of CS 401

Chapter 1

Software Engineering

The field devoted to the **specification, design, production, and maintenance** of non-trivial software products. Includes supporting activities such as

- cost estimation
- documentation
- team organization
- use of tools

The discipline devoted to the **design, production and maintenance** of computer programs that are developed **on time** and **within cost estimate**, using tools that help to **manage the size and complexity** of the resulting software products.

Goals of Quality Software

- It works.
- It can be modified without excessive time and effort.
- It is reusable.
- It is completed on time and within budget.

Life Circle of Software

- Problem analysis
- Requirements elicitation
- Software Specification
- High and low design
- Implementation
- Testing and verification
- Delivery
- Operation
- Maintenance

Object Orientation

Objects represent

- information: we say the objects have attributes.
- behavior: we say the objects have responsibilities.

Objects can represent "real-world" entities such as bank accounts. Objects are self-contained and therefore easy to implement, modify, and test for correctness. Object-oriented classes, when designed properly, are very easy to reuse.

Classes, Objects, and Applications

- An object is an instantiation of a class.
- Alternately, a class defines the structure of its objects.
- A class definition includes variables (data) and methods (actions) that determine the behavior of an object.

Data Structures

A language's set of primitive types are not sufficient, by themselves, for **dealing with data that have many parts and complex interrelationships among those parts**. Data structures provide this ability.

Chapter 2

Data

The **representation** of **information** in a manner suitable for **communication** or **analysis** by **human** or **machine**.

Data type

A category of data characterized by the supported elements of the category and the supported operations on the elements.

1. Primitive types

   A data type whose elements are single non-decomposable data items.

2. Composite types

   A data type whose elements are composed of multiple data items.

   - **Structured composite type**
   - **Unstructured composite type**

Data Abstraction

The separation of a data type's **logical properties** from its **implementation**.

Data Encapsulation

The separation of the representation of data from the application that use the data at a logical level.

Abstraction

A model of a system that includes only the details **essential to the perspective** of the **viewer** of the system.

Information Hiding

The Practice of hiding details within a module with the goal of controlling access to the details from the rest of the system.

Abstract Data Type

A data type whose **properties are specified independently** of any particular **implementation**.

ADT Perspectives or Levels

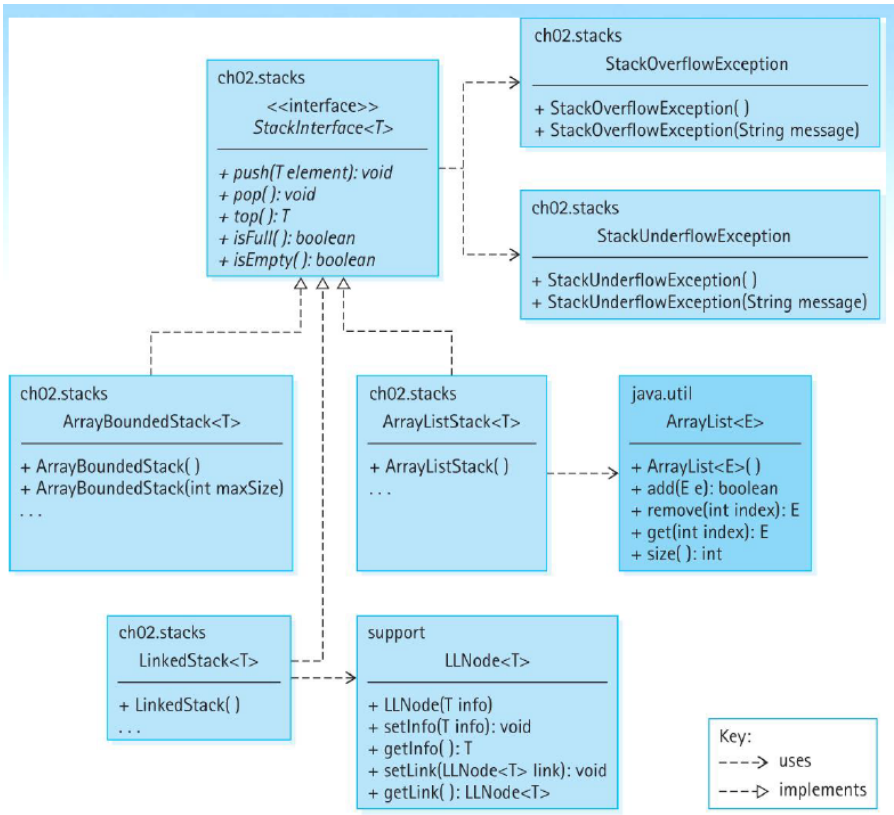
- Application Level
- Logical Level
- Implementation Level

Abstract Method

- Only include a description of its parameters.
- No method bodies or implementations are allowed.
- In other words, only the interface of the method is included.

Interface and UML

- Similar to a Java class
    - can include variable declarations: must be constants(final variables).
    - can include methods: must be abstract.
    - no constructor.
- It provides a template for classes to fill.
- Benefits from using Interfaces
    - Check syntax of our specification
    - verify the interface "contract" is met by the implementation



Chapter 3

Recursion Definition

A definition in which something is **defined** in terms of **smaller version of itself**.

Recursive Algorithms

A solution that is expressed in terms of

- a **smaller instance of itself**
- a **base case**

Base Case

The case for which the solution can be stated **non-recursively**

General Case

The case for which the solution is expressed in terms of **a smaller version of itself**.

Recursive Call

A method call in which the method **being called** is **the same as** the one **making the call**.

3 Questions for Verifying Recursive Algorithms

1. The Base Case Question

   Is there a **non-recursive** way out of the algorithm, and does the algorithm work correctly for this base case.

2. The Smaller Case Question

   Does each recursive call to the algorithm **involves a smaller case** of the original problem, **leading inescapably to the base case**.

3. The General Case Question

   Assumming the recursive call to the smaller case works correctly, does the algorithm **work correctly for the general case**.

Steps for Designing Recursive Solutions

1. Definition

   Get an **exact definition** of the problem to be solved.

2. Size

   Determine the **size** of the peoblem to be solved on this call to the method.

3. Base Case

   Identify and solve the base case in which the problem can be **expressed non-recursively**.

4. General Case

   Identify and solve the general case correctly in terms of a smaller case of same problem, a recursive call.

Removing Recursion

- Iteration
- Stacking

Chapter 4

Comparing Queue Implementations

Operation efficiency: All operations, for each approach, are O(1). Except for the Constructors: Array-based: O(N), Link-based: O(1)

Chapter 5

Collections ADT

Provide the function of **accessing data based on content**, regardless of the order.

Support **addition, removal and retrieval** of elements.

Allow duplicate elements, not allow $null$ elements.

add and $remove$ operations return a $boolean$ indicating success.

Helper function

- protected access
- use sequential search, O(N).
- sets instance variables $found(boolean)$ and $location(int)$
- simplifies
    - remove
    - contains
    - get

Array Collection and Sorted-array Collection

Array-base collection features fast $add(O(1))$ but slow
$get,\ contains\ and\ remove(O(1))$

Many applications require fast retrieval($get,\ contains$), where we should use sorted-array collections, which employees Binary Search for $find$.

By Copy & By Reference

By Copy

- Created by $clone$ method
- classes must implements $Cloneable$ interface
- drawbacks
    - not up-to-date
    - slow
    - more storage space

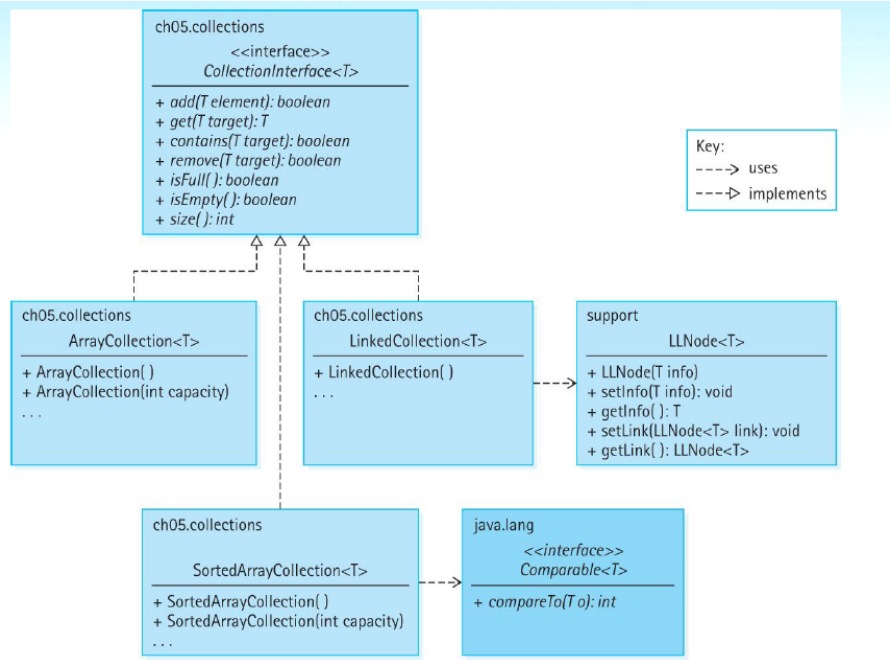
By Reference

- drawbacks
    - carefully deal with aliases.

Which is better

- It depends.
- In big scale, when time and space are problems, use by reference.
- In small scale, when list of objects is not too large, use by copy.

Collections UML



Difference from Stack and Queue

Stack and queue are based on order.

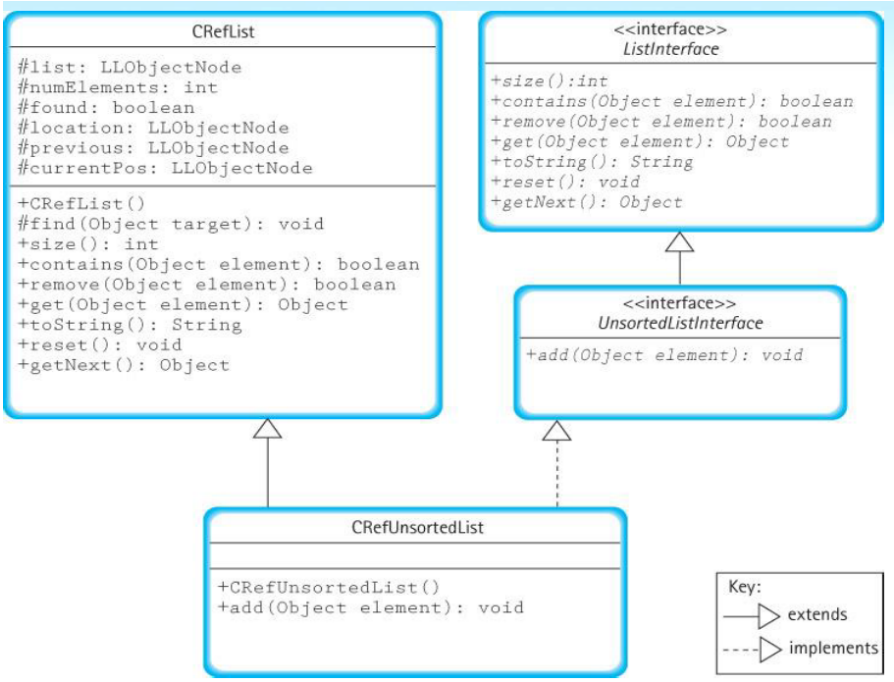
Collection is based on content.

Chapter 6A

Circular Linked Lists

Last element is succeeded by the first element.

Unsorted Circular Linked List UML



Array Approach of Linked List

- Sometimes managing free space ourselves give us greater flexibility
- Some languages doesn't support dynamic allocation (no linked list can be used).
- Sometimes dynamic allocation is too costly.

This array is bounded. You can enlarge it.

Boundedness:

- Desire for static allocation
- Our list will not grow
- Check whether our list is full

Chapter 7 BST

Well balanced BT

Always maintain triangle. Have best performance. Shape of BT is important.

Tree

Definition:

- Tree: A structure with a unique starting node(the root), in which each node is capable of having multiple successor nodes(its children), and in which a **unique path exists from the root to every other node**.

- Root: The top node of a tree structure; a node with no parent.
- Parent node: The predecessor node of a node is its parent.
- Subtree: A node and all of its descendants form a subtree rooted at the node.
- Ancestor: A parent of a node, or a parent of an ancestor.
- Descendant: A child of a node, or a child of a descendant.
- Leaf: A node that has no children.
- Interior node: A node that is not a leaf. Root may not be a interior node.
- Sibling: Nodes with the same parent.
- Level: The level of a node is its distance from the root.
- Height: The maximum level of the tree.

Binary Search Tree

These trees facilitate searching for an element.

All values in the left subtree are less than or equal to the values in the root node.

All values in the right subtree are greater than the value in the root node.

(These properties are called "binary search tree" property.)

A Special Kind of BT

- Each node contains a distinct data value
- The key values in the tree can be compared using "greater than" and "less than", and
- The key value of each node in the tree is less than every key value in its right subtree, and greater than every key value in its left subtree.

Traverse Manner

- Preorder: root, left, right: if added to a new tree, it will be the same tree
- Inorder: left, root, right: to sort and balance
- Postorder: left, right, root

BST Interface

- similar to sorted lists
- extend *CollectionInterface*
- extend *Iterable*
- unbounded, allow duplicate, disallow *null*
- support max and min operation

- support preorder, inorder and postorder traversals

size(): Recursion or Iteration

recursion:

```
private int recSize(BSTNode<T> node)
// Returns the number of elements in subtree rooted at node.
{
  if (node == null)
    return 0;
  else
    return 1 + recSize(node.getLeft()) + recSize(node.getRight());
}
```

iteration:

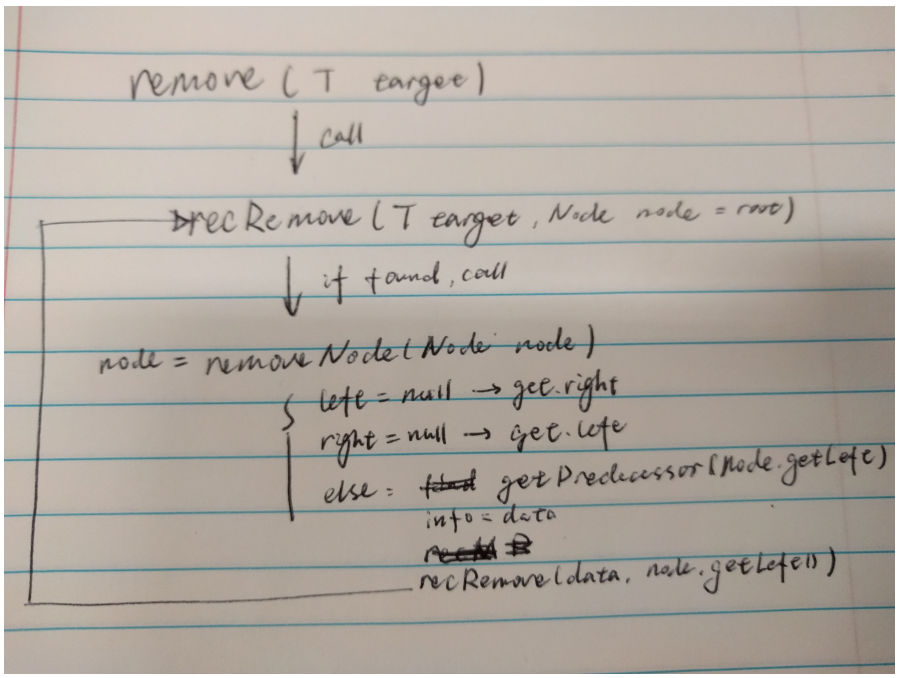
Use stack.

Recursion implement is a good use in this:

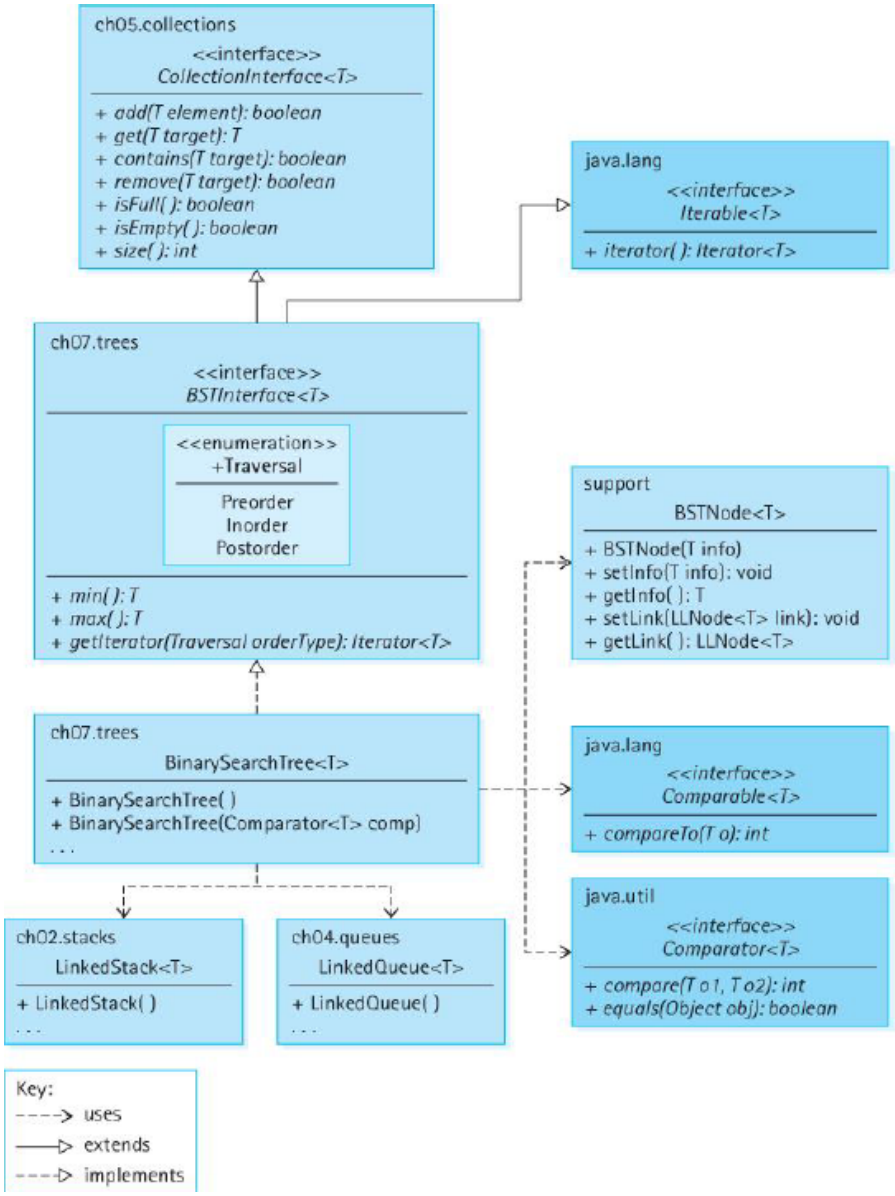
- depth is relatively shallow
- shorter and cleaner
- not much less efficient than iteration

remove()

Remove is the most complicated function in BST



BST UML



Performance

- search quickly
- allows insertions and removals without having to move large amounts of data
- takes more memory space than a singly linked list

BST Big-O

	Binary Search Tree	Array-Based Linear List	Linked List
Class constructor	O(1)	O(N)	O(1)
`isEmpty`	O(1)	O(1)	O(1)
`reset`	O(*N*)	O(1)	O(1)
`getNext`	O(1)	O(1)	O(1)
`contains`	O($\log_2 N$)	O($\log_2 N$)	O(*N*)
`get`			
Find	O($\log_2 N$)	O($\log_2 N$)	O(*N*)
Process	O(1)	O(1)	O(1)
Total	O($\log_2 N$)	O($\log_2 N$)	O(*N*)
`add`			
Find	O($\log_2 N$)	O($\log_2 N$)	O(*N*)
Process	O(1)	O(*N*)	O(*N*)
Total	O($\log_2 N$)	O(*N*)	O(*N*)
`remove`			
Find	O($\log_2 N$)	O($\log_2 N$)	O(*N*)
Process	O(1)	O(*N*)	O(1)
Total	O($\log_2 N$)	O(*N*)	O(*N*)

- **Array-based linear list can employ binary search to get higher performance for *Find***
- ***reset*** of BST is O(N)
- Process of ***remove*** of BST is O(1)

Balance

Basic Algorithm

- save the tree information (**in order**) in an array
- insert the information from the array back (**insert "the middle" recursively**) into the tree

In the insert function, there are three cases: one node, two node, else.

Balance

```
Iterator iter = tree.getIterator(Inorder )
int index = 0
while (iter.hasnext())
    array[index] = iter.next( )
    index++
tree = new BinarySearchTree()
tree.InsertTree(0, index - 1)
```

InsertTree(low, high)

```
if (low == high)                  // Base case 1
   tree.add(array[low])
else if ((low + 1) == high)    // Base case 2
   tree.add(array[low])
   tree.add(array[high])
else
   mid = (low + high) / 2
   tree.add(array[mid])
   tree.InsertTree(low, mid – 1)
   tree.InsertTree(mid + 1, high)
```

Chapter 8 Map

Maps associate a key exactly one value.

- Not permit duplicate keys
- allow multiple keys map onto the same value
- keys can **not** be ***null***; values can be ***null***.

Interface

- put: add/update a pair
- get: put in a key, return the value
- remove: put in a key, return the pair
- contain

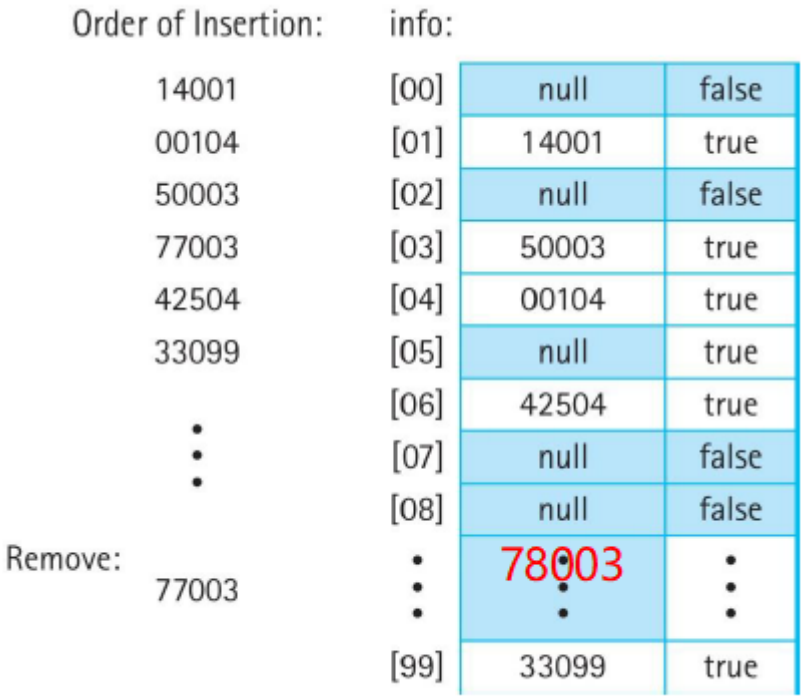
Implementations

	UNSORTED ARRAY	SORTED ARRAY	SORTED LL	UNSORTED LL	BST
put	O(N)	O(N)	O(N)	O(N)	O($\log_2 N$)
get	O(N)	O($\log_2 N$)	O(N)	O(N)	O($\log_2 N$)
remove	O(N)	O(N)	O(N)	O(N)	O($\log_2 N$)
contain	O(N)	O($\log_2 N$)	O(N)	O(N)	O($\log_2 N$)

Hashing

- Hashing is an array, usually called as hash table.
- Typically provides O(1) implement.
- Efficient approach for map.
- Array(hash table) indices are determined directly from kay.

Removal

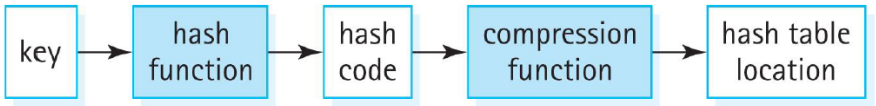


If the boolean value is true, continue searching.

Collision resolution policies

- linear probing: may lead to inefficient cluster of entries.
- quadratic probing
- buckets
- chaining(good)

Steps



key: original key

hash function: convert key to integer with good "spread"

hash code: converted key

compression function: convert hash code to index, such as "%".

hash table location: converted hash code (index of table)

Hashing function

- selecting: try to convert keys to various results
- digitizing: must transform to integer
- combing: build a formula for this

Considerations

- hash code is not unique, is not a key
- same entry have same hash
- think the workload for converting
- the Big-O is between O(1) and O(N). usually not difficult to achieve O(1)

Chapter 9 PQ

Implementation

- Unsorted List
- Array-based Sorted List
- Sorted Linked List
- Binary Search Tree
- Heap

Heap

- shape: a complete tree
- order: node value $\geq$ children value

dequeue: switch root with rightmost leaf, reheapDown the root

enqueue: add to rightmost place, reheapUp the new node

implementation: array

Big-O

	enqueue	dequeue
Heap	$O(\log_2 N)$	$O(\log_2 N)$
Linked List	$O(N)$	$O(1)$
Binary Search Tree		
Balanced	$O(\log_2 N)$	$O(\log_2 N)$
Skewed	$O(N)$	$O(N)$

Chapter 10 Graph

Definitions

- Graph: consist of a set of nodes and a set of edges that relate the nodes to each other
- Vertex: node
- Edge: connection between two nodes
- Undirected graph
- Directed graph

a graph G is defined as follows:

G = (V,E)

where

V(G) is a finite, nonempty set of vertices

E(G) is a set of edges (written as pairs of vertices)

- Adjacent vertices: connected nodes
- Path: sequence of nodes that connects tow nodes
- Complete graph
- Weighted graph

Interface

isEmpty(): boolean

isFull(): boolean

addVertex(T vertex): void

hasVertex(T vertex): boolean

addEdge(T fromVertex, T toVertex, int weight): void

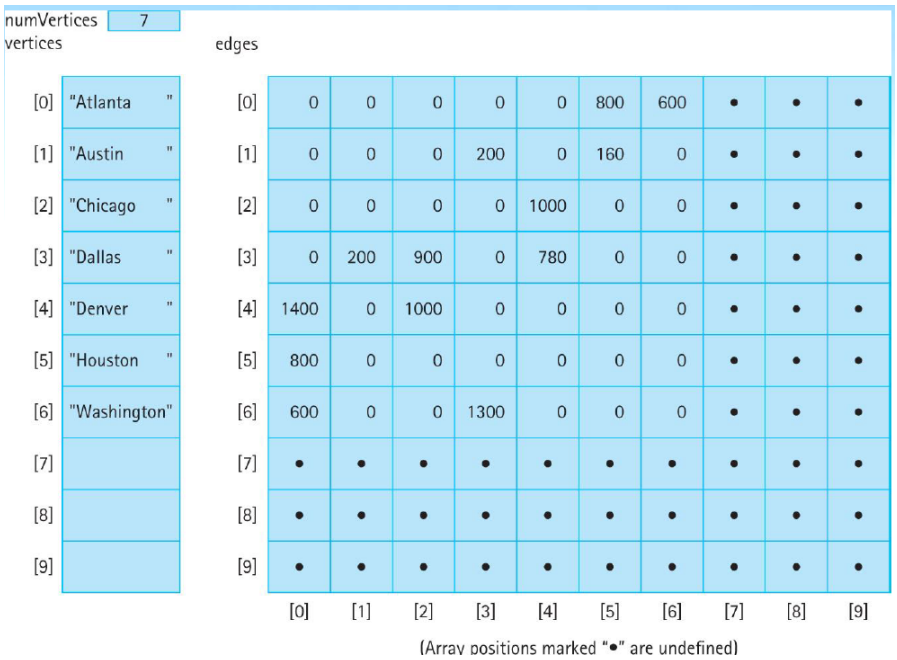
weightIs(T fromVertex, T toVertex): int

getToVertex(T vertex)

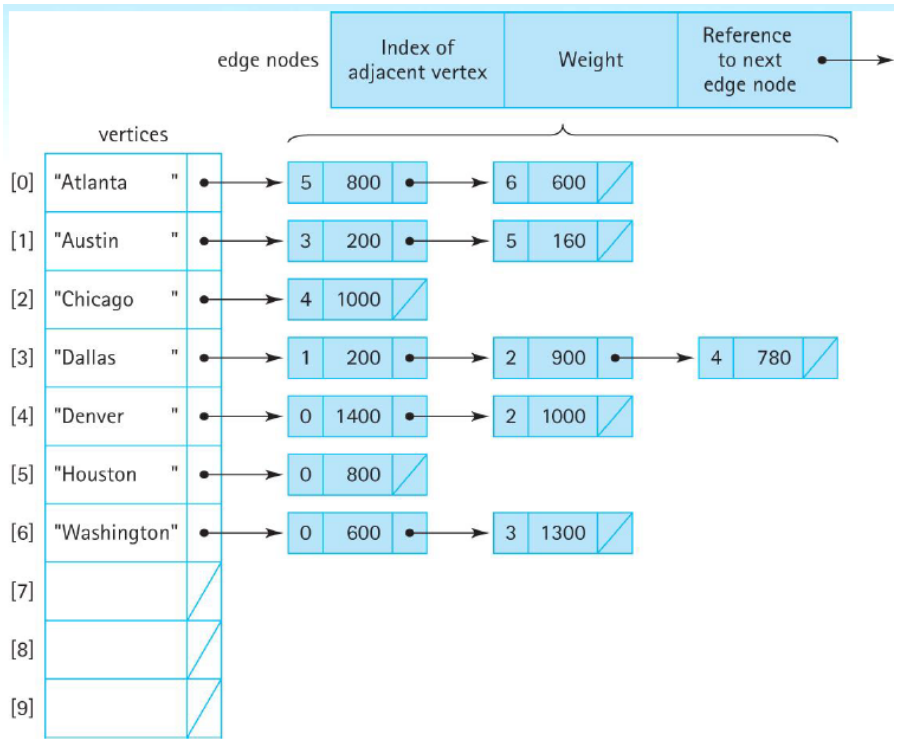
Implementations

- Array-based

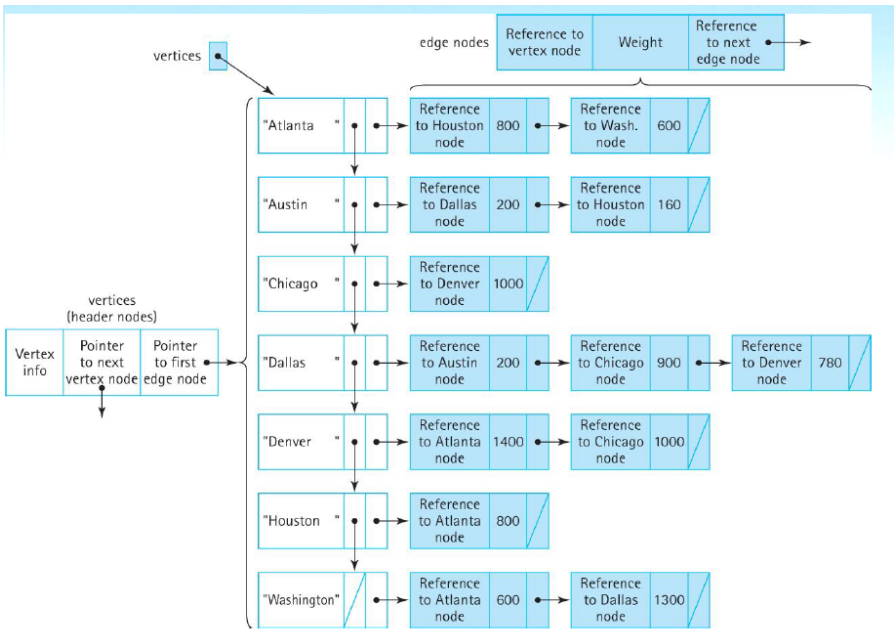
– an integer variable `numVertices`
– a one-dimensional array `vertices`
– a two-dimensional array `edges` (the adjacency matrix)



- Linked



the linked list contains the index of vertex

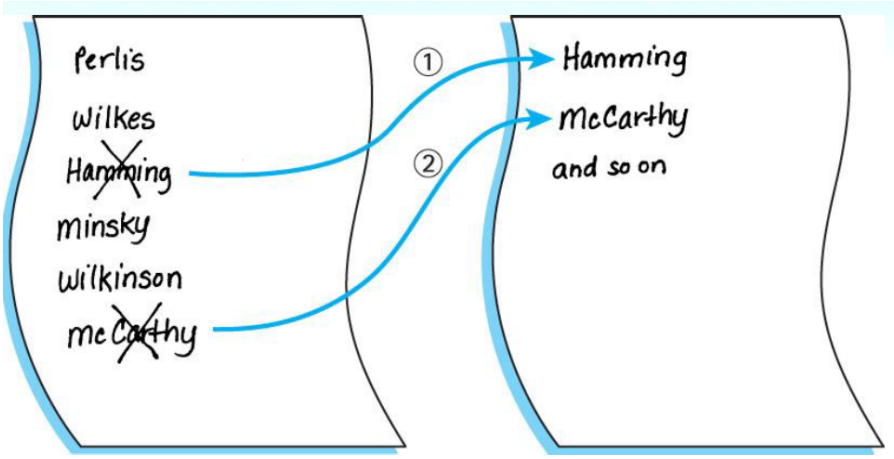


the linked list contains the reference of vertex

Chapter 11 Sorting and Searching

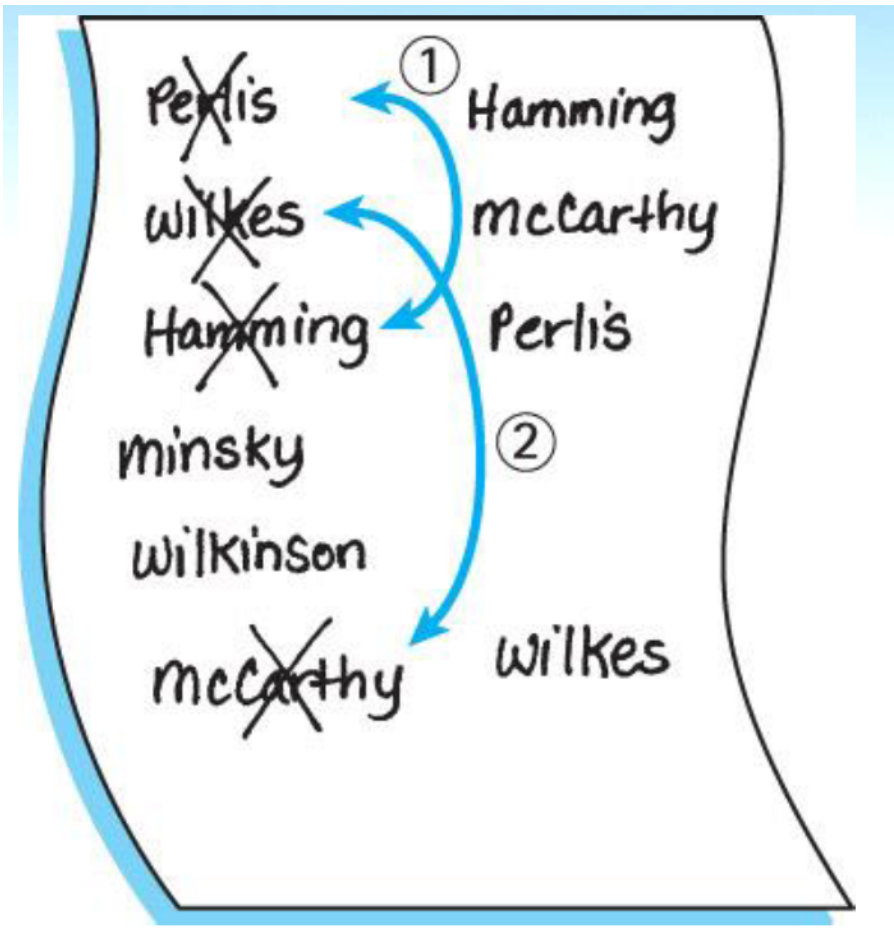
Simple Sorts: O(N) Complexity

Selection Sort



require space to store two lists

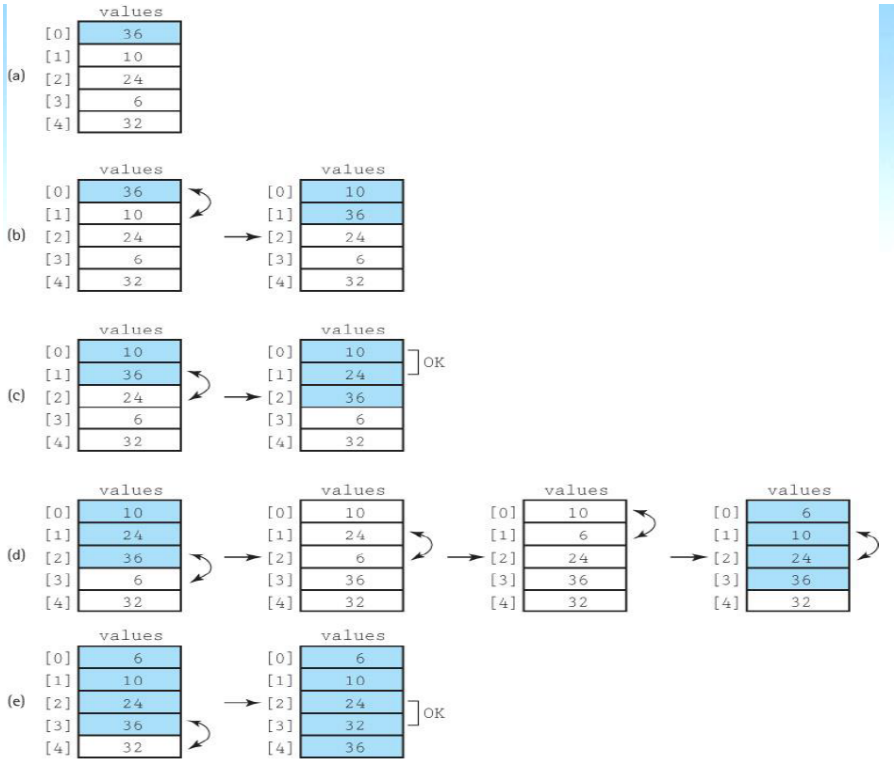
improvement:



Bubble Sort

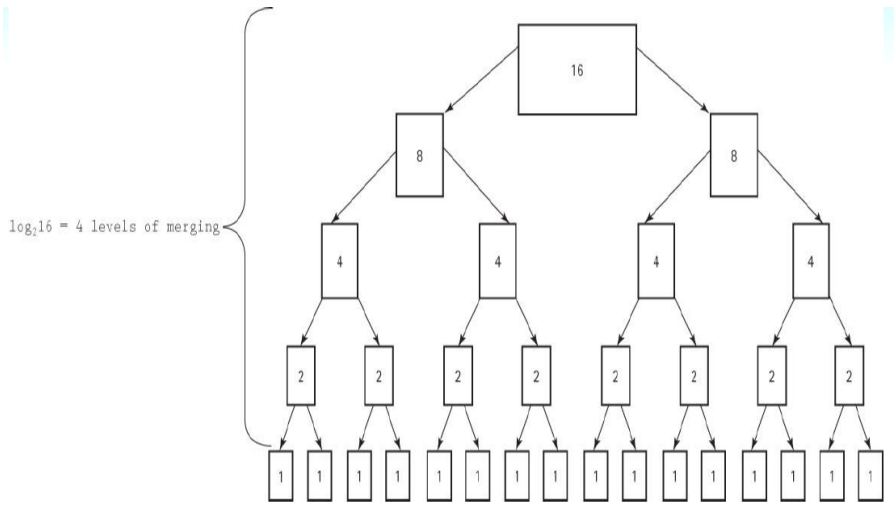
Insertion Sort

Like a deck



$O(N \log_2 N)$ Sorts

Merge Sort

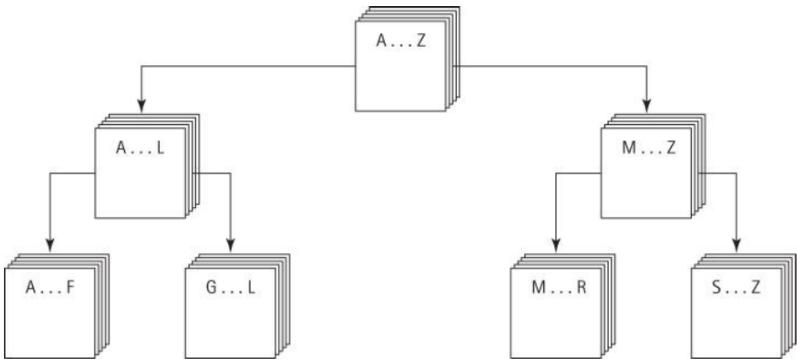


Big-O: splitting=O(N), merging=levels($\log_2 N$)*merging in every level(O(N))=O($N \log_2 N$)

In every level, merging need $2^n(\frac{N}{2^n} - 1)$ of compare.

Drawbacks: need aux space to store a sorted array, so doesn't fit for large array, if space is a problem.

Quick Sort



Split value: we can choose the first value

Big-O: Comparison at each level(O(N)) * number of levels(O($\log_2 N$))=O($N \log_2 N$)

Drawbacks: sensitive to the order of data, the choice of splitting value; aux space for recursion

Good for large scale random data

Heap Sort

Not affected by order of data. Efficient in space. Good choice for large scale data.

```
static void heapSort()
// Post: The elements in the array values are sorted by key
{
  int index;
  // Convert the array of values into a heap
  for (index = SIZE/2 - 1; index >= 0; index--)
    reheapDown(values[index], index, SIZE - 1);

  // Sort the array
  for (index = SIZE - 1; index >=1; index--)
  {
    swap(0, index);
    reheapDown(values[0], 0, index - 1);
  }
}
```

Only to heap down (size/2 - 1)

Good for **large arrays** rather than small arrays.

Isn't affected by order.

Only require constant extra space.

Big-O

Table 11.3 Comparison of Sorting Algorithms

	Order of Magnitude		
Sort	**Best Case**	**Average Case**	**Worst Case**
`selectionSort`	$O(N^2)$	$O(N^2)$	$O(N^2)$
`bubbleSort`	$O(N^2)$	$O(N^2)$	$O(N^2)$
`shortBubble`	$O(N)$*	$O(N^2)$	$O(N^2)$
`insertionSort`	$O(N)$*	$O(N^2)$	$O(N^2)$
`mergeSort`	$O(N \log_2 N)$	$O(N \log_2 N)$	$O(N \log_2 N)$
`quickSort`	$O(N \log_2 N)$	$O(N \log_2 N)$	$O(N^2)$ (depends on split)
`heapSort`	$O(N \log_2 N)$	$O(N \log_2 N)$	$O(N \log_2 N)$

*Data almost sorted.