DataStructure笔记

Data Structure

数据结构

1. Overview

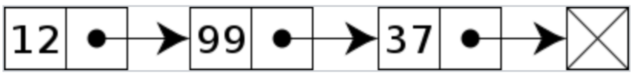
In computer science, a **data structure** is a data organization, management, and storage format that enables efficient access and modification.
More precisely, a data structure is a collection of data values, the relationships among them, and the functions or operations that can be applied to the data.
In this course, we learned several commonly used data structures and sorting algorithms.
Data structures include: linked list, stack, queue, tree, graph.

2. Linked List

In computer science, a linked list is a linear collection of data elements whose order is not given by their physical placement in memory. Instead, each element points to the next. It is a data structure consisting of a collection of nodes which together represent a sequence.

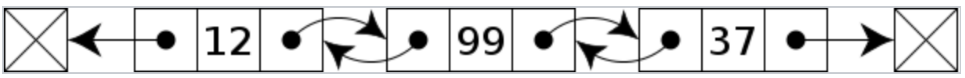
2.1 Singly linked list

Singly linked lists contain nodes which have a **data field** as well as **’next’ field**, which points to the next node in line of nodes.



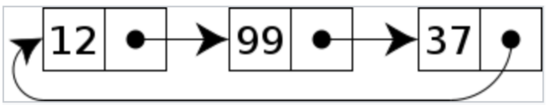
2.2 Double linked list

In a ’doubly linked list’, each node contains, besides the next–node link, a second link field pointing to the ’previous’ node in the sequence.
The two links may be called ’**next**’ and ’**prev**’(’previous’).



2.3 Circular linked list

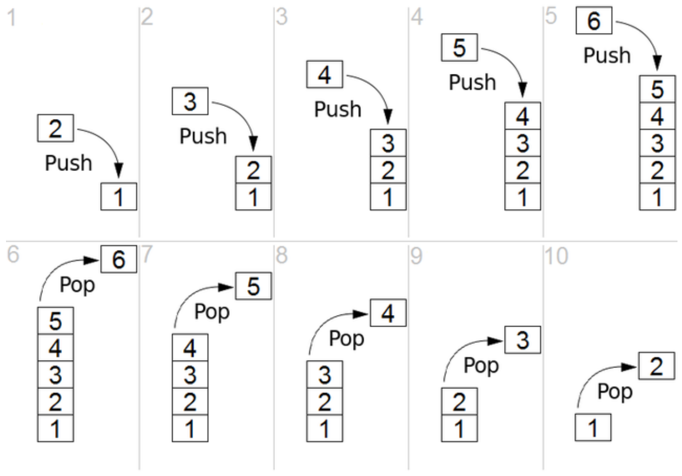
It is a list where the last pointer points to the first node.



In the case of a circular doubly linked list, the first node also points to the last node of the list.

3. Stack

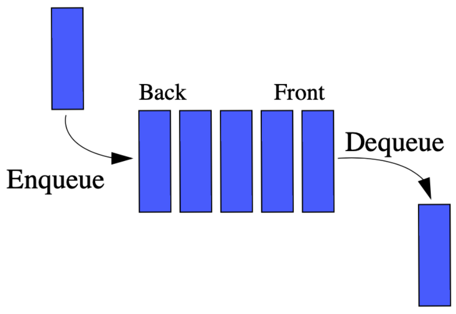
A **stack** is a linear list, it has two operations: **pop (deletion)** and **push (insertiong)**. The push and pop operations occur only at one end of the list, referred to as the **top** of the stack.
A stack is a linear list of **last-in-first-out** (LIFO) structures.



Example:
Labyrinth solution: Every step you walk is a push of stack, if you encounter a dead end, you will pop of stack step by step, and go back to the fork to change a path.

4. Queue

The queue is a **first-in-first-out** (FIFO) data structure. In a FIFO data structure, the first element added to the queue will be the first one to be removed.
The operation of adding an element to the **rear** of the queue is known as **enqueue**, and the operation of removing an element from the **front** is known as **dequeue**.



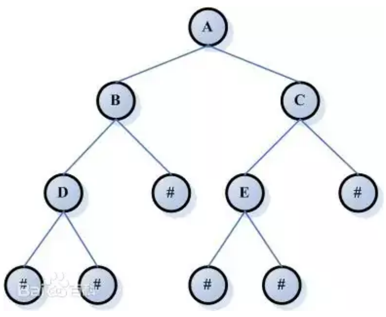
Example:
When a bank handles business, each window will **dequeue** the current customer from the **front** when his business finished, and call the next customer in line behind him.

5. Tree

5.1 Concept

A tree data structure can be defined recursively as a collection of nodes, where each node is a data structure consisting of a **value** and **a list of references to nodes**. The start

of the tree is the "**root node**" and the reference nodes are the "children". No reference is duplicated and none points to the root.
Each node in a tree has zero or more **child nodes**. A node that has a child is called the child's **parent node** (or superior). A node has at most one parent, but possibly many **ancestor nodes**, such as the parent's parent. Child nodes with the same parent are **sibling nodes**.

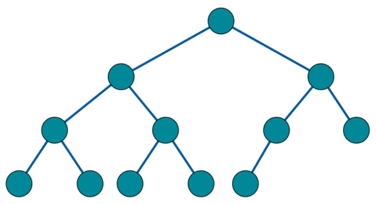


Other terms used with trees:
Degree: For a given node, its number of children. A leaf has necessarily degree zero.
Degree of tree: The degree of a tree is the maximum degree of a node in the tree.
Distance: The number of edges along the shortest path between two nodes.
Level: The level of a node is the number of edges along the unique path between it and the root node.
Width: The number of nodes in a level.
Breadth: The number of leaves.
Forest: A set of $n \geqslant 0$ disjoint trees.

5.2 Binary Tree

The binary tree is a tree data structure in which each node has at most two children, which are referred to as the **left child** and the **right child**.

- A **full binary tree** is a tree in which every node has either 0 or 2 children.
- A **perfect binary tree** is a binary tree in which all interior nodes have two children and all leaves have the same depth or same level.
- In a **complete binary tree** every level, except possibly the last, is completely filled, and all nodes in the last level are as far left as possible. As the picture shows:



5.3 Tree Traversal

Traversing a tree involves iterating over all nodes in some manner. Because from a given node there is more than one possible next node (it is not a linear data structure), then, assuming sequential computation (not parallel), some nodes must be deferred—stored in some way for later visiting. This is often done via a stack(LIFO) or queue (FIFO). As a tree is a self-referential (recursively defined) data structure, traversal can be defined by recursion or, more subtly, corecursion, in a natural and clear fashion; in these cases the deferred nodes are stored implicitly in the call stack.

- Depth-First Search

  In depth-first search (DFS), the search tree is deepened as much as possible before going to the next sibling. To traverse binary trees with depth-first search, perform the following operations at each node:

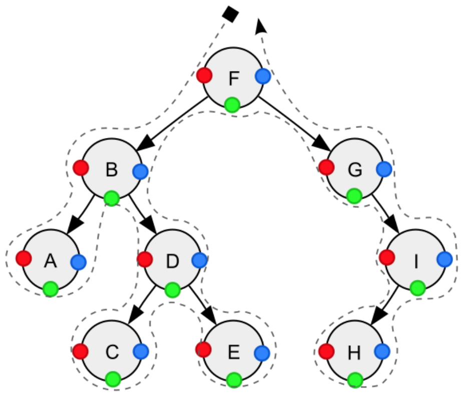
a. If the current node is empty then return.

b. Execute the following three operations in a certain order:

N: Visit the current node.

L: Recursively traverse the current node's left subtree.

R: Recursively traverse the current node's right subtree.



Depth–first traversal (dotted path) of a binary tree:

- Pre–order, NLR (node visited at position red 🔴):

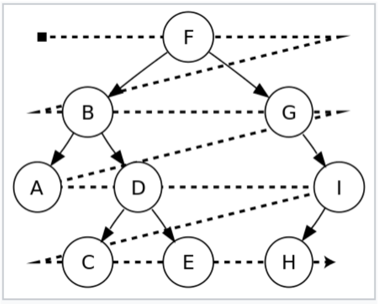
  F, B, A, D, C, E, G, I, H;
- In–order, LNR (node visited at position green 🟢):

  A, B, C, D, E, F, G, H, I;
- Post–order, LRN (node visited at position blue 🔵):

  A, C, E, D, B, H, I, G, F.

- Breadth–First Search

  In breadth–first search (BFS) or level–order search, the search tree is broadened as much as possible before going to the next depth.



Level-order: F, B, G, A, D, I, C, E, H.

6. Graph

相关课程：离散数学

6.1 Concept

A **graph** data structure consists of a finite set of **vertices** (also called nodes or points), together with a set of **unordered pairs** of these vertices for an **undirected graph** or a set of **ordered pairs** for a **directed graph**. These pairs are known as **edges**(also called links or lines), and for a directed graph are also known as edges but also sometimes **arrows or arcs**.

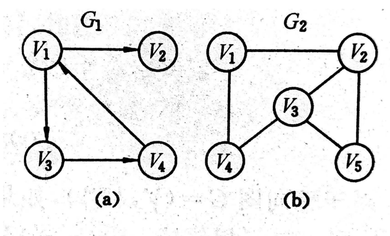


图 7.1　图的示例
(a) 有向图 G_1；(b) 无向图 G_2

6.2 Storage Structure

The graph is represented by a multi-linked list, that is, a vertex in the graph is represented by a node consisting of a **data field** and **multiple pointer fields**, where the data field stores the vertex information, and the pointer field stores pointers to adjacent nodes.

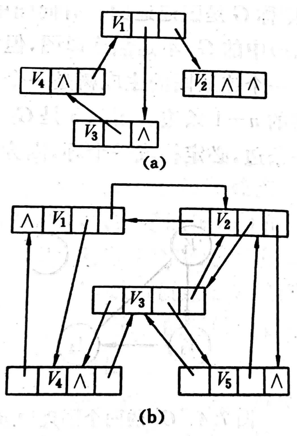


图 7.7　图的多重链表
(a) G_1 的多重链表；(b) G_2 的多重链表

6.3 Graph traversal

Graph traversal refers to the process of visiting each vertex in a graph. Such traversals are classified by the order in which the vertices are visited. Tree traversal is a special case of graph traversal.

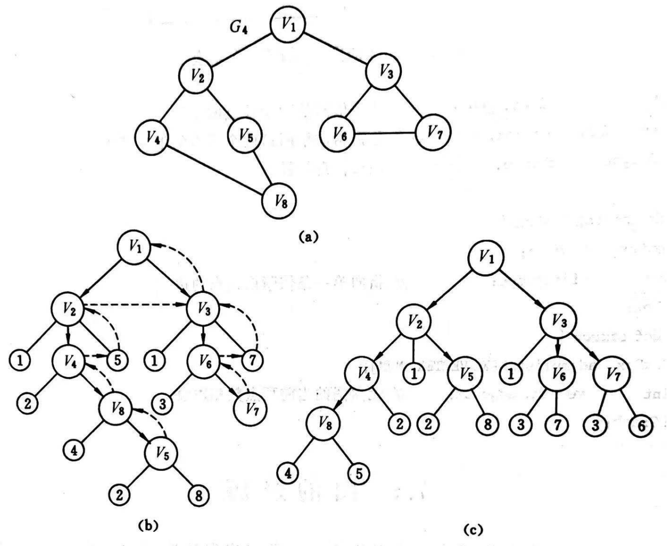


图 7.13　遍历图的过程
(a) 无向图 G_4；(b) 深度优先搜索的过程；(c) 广度优先搜索的过程

- Depth-First Search (DFS):
    - DFS visits the child vertices before visiting the sibling vertices; that is, it traverses the depth of any particular path before exploring its breadth. A **stack** (often the

program’s call stackvia recursion) is generally used when implementing the algorithm.
  - The above example: the access order is v1–v2–v4–v8–v5–v3–v6–v7
- **Broadth-First Search (BFS)**:
  - BFS visits the sibling vertices before visiting the child vertices, and a **queue** is used in the search process.
  - The above example: the access order is v1–v2–v3–v4–v5–v6–v7–v8

The process of traversing the graph is essentially the process of **finding adjacent points through edges or arcs**, so the time complexity of DFS and BFS is the same, only the order of accessing vertices is different.

6.4 Minimum Cost Spanning Tree

A graph data structure may also associate to each edge some edge value, such as a symbolic label or a numeric attribute (cost, capacity, length, etc.). The weights assigned to the edges represent the corresponding costs, and the cost of a spanning tree is the sum of the costs of the edges in the tree. The minimum spanning tree problem is how to construct the **minimum cost spanning tree** of the connected network.

- Pirm algorithm

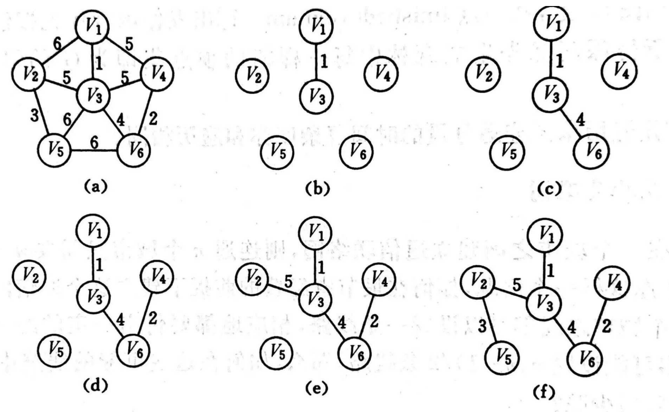


图 7.16 普里姆算法构造最小生成树的过程

- Kruskal algorithm

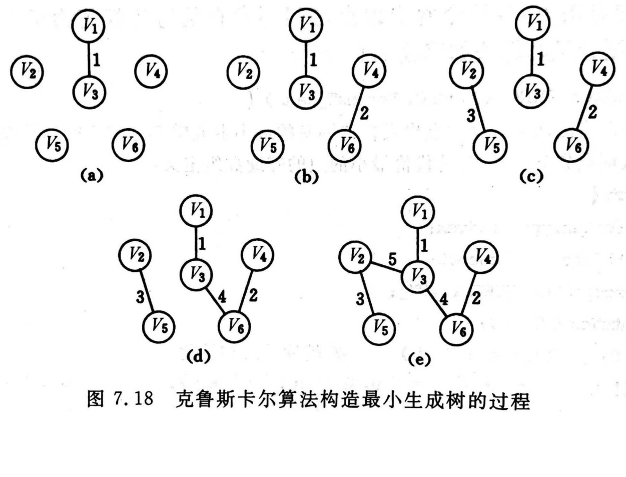


图 7.18 克鲁斯卡尔算法构造最小生成树的过程

7. Sorting

A **sorting algorithm** is an algorithm that puts elements of a list into an order.

7.1 Insertion Sort

Insertion sort is a simple sorting algorithm that is relatively efficient for small lists and mostly sorted lists, and is often used as part of more sophisticated algorithms.

It works by taking elements from the list one by one and inserting them in their correct position into a new sorted list.

7.2 Selection Sort

The algorithm finds the minimum value, swaps it with the value in the first position, and repeats these steps for the remainder of the list.

```
void SelectSort(SqList &L) {
  for(i=1; i<L.length; ++i){
    j = SelectMinKey(L, i);      // 在L中选择从i到尾key最小的记录并返回位置
    if(i!=j) L.r[i]<-->L.r[i];  // 最小记录与i交换位置
  }
}
```

7.3 Bubble Sort

Bubble sort is a simple sorting algorithm. The algorithm starts at the beginning of the data set. It compares the first two elements, and if the first is greater than the second, it swaps them. It continues doing this for each pair of adjacent elements to the end of the data set. It then starts again with the first two elements, repeating until no swaps have occurred on the last pass.

7.4 Quick Sort

Quicksort is a divide and conquer algorithm which relies on a partition operation: to partition an array, an element called a **pivot** is selected. All elements smaller than the pivot are moved before it and all greater elements are moved after it. This can be done efficiently in linear time and in-place. The lesser and greater sublists are then recursively sorted.
Efficient implementations of quicksort (with in-place partitioning) are typically unstable sorts and somewhat complex, but are among the **fastest sorting** algorithms in practice.

```
// 一趟排序
int Partition(SqList &L, int low, int high) {
  pivotkey = L.r[low].key;  // 用第一个记录作为枢轴记录
  while(low<high) {       // 从表的两端交替向中间扫描
    while(low<high && L.r[high].key>=pivotkey) --high;
    L.r[low]<-->L.r[high]; // 将比枢轴记录小的记录交换到低端
    while(low<high && L.r[low].key<=pivotkey) ++low;
    L.r[low]<-->L.r[high]; // 将比枢轴记录大的记录交换到高端
  }
  return low;             // 返回枢轴所在位置
}

// 整个快速排序过程可递归进行
void Qsort(SqList &L, int low, int high) {
  if(low<high) {
    pivotloc = Partition(L, low, high); // 将L一分为二，得到枢轴位置
    Qsort(L, low, pivotloc-1);          // 对低子表递归排序
    Qsort(L, high, pivotloc+1);         // 对高子表递归排序
  }
}

// 对顺序表L做快速排序
void QuickSort(SqList &L) {
  QSort(L, 1, L.length);
}
```

References:
思维导图
Data Structure – Wikipedia
Linked List – Wikipedia
Stack (abstract data type) – Wikipedia
Queue (abstract data type) – Wikipedia
Tree (data structure) – Wikipedia
Binary Tree – Wikipedia
Tree traversal – Wikipedia
Graph (abstract data type) – Wikipedia
Graph traversal – Wikipedia
Sorting algorithm – Wikipedia