ChatGPT

# 資料結構期中考複習筆記

## 1. 鏈結串列（Linked List）

### 基本概念與術語定義

鏈結串列（linked list）是一種動態線性資料結構，由一系列節點（node）串接而成。每個節點包含兩部分：**資料區**（通常稱為`info`，儲存節點的值）和**鏈結區**（通常稱為`link`或`next`，儲存指向下一個節點的指標） [1] [2]。串列開頭的節點位址通常由一個指標變數保存，常命名為`head`、`first`或`list` [3]。當指標為空（`nullptr`或`NULL`）時，表示串列結束或不存在有效節點 [3]。

鏈結串列的長度具有彈性，可動態擴張，不必像陣列那樣在建立時預先定義大小 [4]。但與陣列相比，鏈結串列**只能順序訪問**元素，無法透過索引直接隨機存取 [5]。相反地，陣列支援O(1)的隨機存取及（若已排序時）二分搜尋，但插入和刪除元素須移動大量資料（O(n)時間）且大小固定可能導致溢位問題 [6]。鏈結串列則支援常數時間的插入與刪除（假設已取得操作位置的參考），使用`new`和`delete`動態配置節點記憶體，不會有固定大小限制 [5]。

### 資料結構特性與操作原理

**基本操作**包括： - **初始化與判空**：將`head`初始化為`nullptr`表示空串列；判斷`head`是否為空指標即可檢查串列是否為空。 - **遍歷（Traversal）**：從`head`開始，沿著`link`逐節點走訪至串列結束，用於列印或計算長度等。例如，下列函式計算串列長度，即每走訪一節點計數+1 [7] [8]：

```
int length(const node<Type>* list) {
  int len = 0;
  const node<Type>* p = list;
  while (p != nullptr) {
    len++;
    p = p->link;
  }
  return len;
}
```

- **搜尋（Search）**：從`head`起順序比對節點資料，尋找目標值。時間複雜度為O(n)。 - **插入（Insertion）**：將新節點插入串列，可在頭部、尾部或中間位置。插入時需更新相鄰節點的指標。常見做法是： - **頭插入**：新節點`p`的`link`指向原本的第一節點，然後更新`head`為`p` [9]。由於只需修改常數個指標，頭插入操作為O(1)。 - **尾插入**：需要先找到尾節點（`link == nullptr`），再將其`link`指向新節點。無尾指標時搜尋尾部需O(n)時間，但若維護尾指標則可O(1)完成。 - **中間插入**：需先找到插入位置前一節點`prev`，再調整指標：讓新節點的`link`指向`prev->link`原位置，再讓`prev->link`指向新節點 [10] [11]。若位置在串列開頭，則等價於頭插入特例 [12]。 - **刪除（Deletion）**：移除指定節點。需要注意的是，為了移除某節點`cur`，我們必須取得其前驅節點`prev`以便修改鏈結 [13] [14]。典型算法： 1. 使用一個指標`cur`順序搜尋欲刪除值`x`所在的節點，並記錄其前驅`prev` [15]。 2. 若找到`x`，分情況討論： - 若`cur`不是第一個節點（`prev != nullptr`），將`prev->link`指向`cur->link`跳過`cur` [16]。 - 若`cur`是第一節點（`prev ==`

`nullptr`），表示刪除頭節點，直接將`head`更新為`cur->link` [16]。3. 呼叫`delete cur`釋放節點記憶體 [16]。4. 若找到並刪除了節點，回傳`true`，否則回傳`false` [17]。

```
bool remove(node*& list, int x) {
  node *cur = list, *prev = nullptr;
  while (cur != nullptr && cur->info != x) {    // 尋找值x
    prev = cur;
    cur = cur->link;
  }
  if (cur != nullptr) {                  // 找到值x節點
    if (prev != nullptr) prev->link = cur->link;
    else list = cur->link;               // 刪除頭節點
    delete cur;                          // 釋放節點
    return true;
  }
  return false;
}
```

上述刪除操作除搜尋外只涉及指標調整，效率近似O(1)。但搜尋目標值本身是線性O(n)工作。

### C++實作範例與程式碼說明

在C++中，可以使用`struct`定義鏈結串列的節點類別，例如整數型別的單向鏈結串列節點：

```
struct Node {
  int info;
  Node *link;
};
Node *head = nullptr;  // 指向串列第一個節點的指標（初始為空）
```

上述結構可彈性運用於不同資料型別，或利用模板`template<class Type>`實作泛型節點 [2] [18]。操作函式可分為兩類：不修改串列的**訪問函式**（getter）及會修改串列結構的**更新函式**（setter）。例如：- **getter範例**：`getMin(node *list)`遍歷找出串列中最小值 [19] [20]；`getMinNode(node *list)`則回傳最小值所在的節點指標 [21] [22]。- **setter範例**：`removeMin(node*& list)`移除串列中資料最小的節點並回傳其值 [23]；`insert(node*& list, node *newNode)`將`newNode`依大小順序插入有序串列的正確位置 [24] [25]。

另一個實作重點是**特殊鏈結串列變體**：　-　**帶**`**表頭節點**`的串列：在串列開頭增加一個不存放有效資料的dummy節點，可簡化某些插入/刪除邊界條件。 - **雙向鏈結串列（Doubly Linked List）**：每個節點有兩個鏈結，`next`指向後繼，`prev`指向前驅 [26]。雙向串列可以雙向遍歷，且在已知節點位置時，可在O(1)時間內刪除該節點（因為能直接取得前驅節點，不用額外搜尋） [27]。插入時也因為有前驅資訊而更方便。代價是每個節點多儲存一個指標且維護額外指標成本。 - **環狀鏈結串列（Circular Linked List）**：尾節點的`next`指回頭節點，使整個串列形成環狀。常用於需要循環訪問的情境，例如實現圓桌順序處理等。要注意避免無窮循環遍歷，需要特別的迴圈終止條件（通常是記錄起點再次被訪問）。 - **上述結合**：也有**環狀雙向串列**、**帶表頭的雙向環狀串列**等形式，在特殊應用中使用（例如作業系統的行程管理常採用環狀雙向串列加表頭以方便插入刪除）。

### 時間與空間複雜度分析

對於單向鏈結串列，各種操作的時間複雜度概括如下： - **搜尋**：O(n)。需線性掃描最壞需遍歷全部n個節點。 - **取得第k個節點**：O(n)。不支援隨機索引訪問，需從頭順走k步。 - **插入**：在已知插入點（或前驅節點）的前提下是O(1)，因為只是修改常數個指標。然而如果按值插入有序串列，需要先O(n)搜尋插入位置，總體變為O(n)。 - **刪除**：已知待刪節點的前驅則指標調整O(1)；若按值刪除需先搜尋O(n)。 - **長度計算**：如果不維護長度變數，需遍歷計算O(n)。可以選擇每次插刪時更新計數器以O(1)換取即時長度查詢。

空間方面，每個節點除了存放數據還需額外一個指標空間。相對陣列，鏈結串列節點間存儲不連續，有指標開銷，但優點是節點數可按需動態增減，不會浪費或耗盡固定空間。

### 可能的錯誤與除錯要點

- **空指標錯誤**：在訪問節點指標成員前，一定確認指標非`nullptr`[28]。否則可能解引用空指標導致執行時錯誤（Null Pointer Exception）。
- **遺漏更新`head`或指標串接**：插入或刪除第一個節點時，需特別處理`head`的變動。如果刪除了頭節點但忘記更新`head`，會導致串列遺失開頭。插入到空串列時，`head`也需更新指向新節點。
- **遺漏釋放記憶體**：動態分配的新節點在不需要時要`delete`釋放，否則造成記憶體洩漏。特別是批量刪除（如清空串列）時，應逐節點刪除並釋放。
- **錯誤地改動節點次序**：例如在刪除節點時，如果不小心調換了前驅和後繼的更新順序，可能造成串列斷鏈或遺失部分節點。調試時可打印節點序列確認鏈結是否保持正確。
- **指標別名問題**：當有多個指標引用串列節點時，改動其中一個需要注意同步（例如兩個指標指向同一節點時，一個刪除了節點，另一個就變成野指標）。

### 示意圖與補充說明

由於鏈結串列透過指標將節點連接成序列，可視覺化理解如下：每個節點畫成一個盒子，分成資料欄與指標欄。指標欄箭頭指向下一個節點的位置，最後一個節點的指標箭頭指向`NULL`（空位）。插入節點時，可想像**重新接線**：讓新節點的箭頭接上原後續節點，再讓前驅節點的箭頭改指新節點；刪除時則繞過要刪的節點直接連起前後節點，然後移除該節點。這些操作都只涉及局部的指標調整，但務必小心順序以免斷開鏈結。

---

## 2. 堆疊（Stack）

### 基本概念與術語定義

堆疊（stack）是一種**後進先出（LIFO，Last In First Out）**的抽象資料型別和資料結構[29]。它就像一疊盤子，新添加的元素放在堆疊頂端（top），移除時也是從頂端取走最新加入的元素。由於插入和刪除僅發生在同一端（頂端），因此堆疊具有「後進先出」的特性。

**常用操作**包括： - **初始化**`initialize()`：將堆疊狀態重置為空。 - **判斷空**`empty()`：檢查堆疊是否沒有任何元素[30]。 - **進堆疊**`push(x)`：將元素`x`壓入堆疊頂端[31]。 - **出堆疊**`pop()`：將頂端元素彈出並移除[31]。 - **讀取頂值**`top()`：取得堆疊頂端元素的值（通常不移除該元素）。 - **判斷滿**`full()`：若使用固定大小的底層容器（如陣列）實作，可能需要檢查是否已達容量上限[32]（但若用動態結構則不一定需要此函式[33]）。 - **元素數量**`size()`：返回堆疊中目前元素個數[34]。

### 資料結構特性與操作原理

堆疊的LIFO特性決定了其操作**只能在頂端進行**。這帶來許多應用上的方便，特別是在**遞迴**與**運算式處理**等場合。典型應用例如： - **函式呼叫堆疊**：系統運行時使用堆疊保存函式的區域變數和返回位址，每次呼叫函式時將

資料壓入堆疊，函式返回時彈出 [35] [36]。這解釋了遞迴呼叫的運作原理：每層遞迴對應一個堆疊訊框（stack frame）。 - **手動模擬遞迴**：利用明確的 `stack` 容器，我們可以將遞迴算法改寫為迭代。例如，一個遞迴函式 `what1_rec(x)` 先輸出 `x` 再遞迴調用 `what1_rec(x-3)` 和 `what1_rec(x/2)` [37]。若要改用疊代實作，可用一個棧模擬呼叫流程：先將初始參數壓棧，然後以迴圈彈出參數並處理，過程中按需要將遞迴分支的參數再壓入堆疊 [38]。這種技巧在複雜遞迴（如深度優先搜尋）中很有用。 - **表達式求值與轉換**：堆疊可用來實現中序轉後序（infix to postfix）轉換或直接計算後序表示式。在轉換中使用一個運算子堆疊來暫存尚未輸出的運算子，依據運算子優先級決定何時輸出 [39]。計算後序時，則使用一個操作數堆疊，讀到數字時壓入，讀到運算子時彈出相應數量操作數計算再壓回。這利用了堆疊先處理最近的未配對符號或運算的特性。

**實作方面**，堆疊通常有兩種主要底層結構： - **陣列實作**：用一個固定大小或動態可調的陣列來存放元素，並用一個索引（`stackTop`）指向目前頂端位置 [40]。初始化時 `stackTop = -1` 表示空堆疊 [41]。`push` 操作將 `stackTop` 加1並放入新值，`pop` 操作則讀取頂值後將 `stackTop` 減1 [42] [43]。需注意陣列實作可能**溢位**（overflow）：當 `stackTop` 達到陣列最大索引時，再push就超出容量。因此可在 `push` 前檢查 `full()` [32]；或者如教材建議的擴充策略，當陣列滿時自動倍增容量重新配置 [44] [45]。同樣地，對空堆疊執行 `pop()` 會造成**下溢**（underflow），實作時應檢查避免或拋出異常 [46]。 - **鏈結串列實作**：用單向鏈結串列將每次push的新節點插入串列頭部作為棧頂（這樣pop只需移除頭節點）。由於鏈結串列可動態配置節點，此實作理論上不存在容量限制，不需要full()函式 [33] [47]。STL的 `std::stack` 預設即以 `deque`（雙端佇列）為底層，可視為同時支持頭尾進出的雙向串列，因此不太需要顧慮容量 [48]。

### C++實作範例與程式碼說明

C++提供了現成的 `<stack>` 容器類別，我們可以直接使用：

```
#include <stack>
stack<int> s;
s.push(10);
int x = s.top();  // 讀取棧頂（x將得到10）
s.pop();      // 彈出棧頂元素
```

如果自己實作，可參考以下簡化的陣列堆疊類別介面設計：

```
template<class T>
class Stack {
private:
  int maxSize;
  int stackTop;
  T *list;
public:
  Stack(int size=100) : maxSize(size), stackTop(-1) {
    list = new T[maxSize];
  }
  ~Stack() { delete[] list; }
  bool empty() const { return stackTop < 0; }
  bool full() const { return stackTop >= maxSize - 1; }
  void push(const T& item) {
    if (full()) cerr << "Stack overflow\n";
    else list[++stackTop] = item;
  }
  void pop() {
```

```
    if (empty()) cerr << "Stack underflow\n";
    else --stackTop;
  }
  T top() const {
    if (empty()) throw runtime_error("Stack is empty");
    return list[stackTop];
  }
  int size() const { return stackTop + 1; }
};
```

以上`Stack`類別透過陣列`list`保存元素，`stackTop`為棧頂索引。`push`時先檢查是否滿，未滿則將`stackTop`加1後放入元素；`pop`時若不空則只需遞減`stackTop`即可“移除”元素（實際資料可留在陣列中等待覆蓋）。`top()`則返回`list[stackTop]`。需注意在真實實作中，可能會在`push`時遇滿則自動擴容陣列，以提高實用性。

### 時間與空間複雜度分析

堆疊所有主要操作（push, pop, top, empty, size）在理想情況下都是O(1)時間複雜度。因為每次操作都只涉及陣列尾部索引的移動或鏈結串列的局部指標變更，與資料量無關。而空間複雜度取決於元素數量n，一般為O(n)。在陣列實作中，事先配置的最大容量可能比實際元素多一些空間，但攤銷後仍為線性。使用鏈結串列時，每個元素需要額外指標空間。

### 可能的錯誤與除錯要點

- **下溢與上溢**：對空堆疊執行`pop()`或`top()`會導致下溢錯誤，應在調用前以`empty()`確認不空 46 。同理，對有容量上限的堆疊，push前應檢查不滿或正確處理溢出的情況（擴容或報錯） 44 49 。C++ STL的`stack`在這些情況下會透過底層容器的方法（如`std::deque`）自動處理或throw，所以自己實作時要謹慎防範。
- **未初始化頂指標**：自製陣列堆疊易犯錯是忘記將`stackTop`初始化為-1（或0表示空的約定） 41 。沒有正確初始化會使第一次push的位置錯誤或top讀取到垃圾值。
- **記憶體釋放**：如果堆疊類別使用動態分配陣列，不要忘記在析構時`delete[]`以防記憶體洩漏 50 。鏈結串列實作則要在清空堆疊或銷毀時逐節點刪除。
- **調試**：可以借助輸出堆疊內容來調試。例如，每次push/pop後印出棧頂或整個棧內容，確認操作是否正常執行預期的後進先出行為。

### 應用示例與小結

總而言之，堆疊提供了一種受限但高效的後進先出操作結構。它應用廣泛，例如利用堆疊可以很方便地反轉資料順序（因為彈出順序與壓入相反）、檢查程式語法中的括號匹配（遇左括號push，遇右括號pop匹配）等。掌握堆疊對理解遞迴執行和實作某些演算法（如深度優先搜尋、評估後序表達式）非常重要。在程式實作上，多利用STL的`stack`容器可減少錯誤，除非有特殊需求自行實作。

---

## 3. 佇列（Queue）

### 基本概念與術語定義

佇列（queue）是一種**先進先出（FIFO，First In First Out）**的資料結構。可將其比擬為排隊：最先進入佇列的元素最先被移出。佇列有兩個端點：一端是**對尾（rear）**，進行新增元素；另一端是**對頭（front）**，進行移除元素 51 。因此元素在佇列中按進入順序排列，刪除順序與加入順序相同。

**基本操作**包括： - **初始化**：將佇列設為空狀態（`front == rear`指向同一初始位置）。 - **判斷空**`empty()`：檢查佇列是否無任何元素 [52]。- **元素個數**`size()`：返回佇列中元素的數目 [52]。- **入佇列（入隊）**`enqueue(x)`：在對尾插入新元素。STL中對應`queue.push()` [53]。 - **出佇列（出隊）**`dequeue()`：移除對頭元素。STL中對應`queue.pop()` [53]。 - **取對頭**`front()`：取得對頭元素的值（不移除）。STL的`queue.front()`可用。 - （有些實作也提供`back()`以取得隊尾元素值，以及如前述`full()`檢查已滿的方法，但STL的queue在底層可動態擴充，一般不需full判斷 [54]。）

### 資料結構特性與操作原理

佇列的關鍵在於**兩端操作**：進入從尾，移出從頭。這可以用**循環陣列**或**鏈結串列**來實作：- **循環陣列實作**：用一個固定大小的陣列配合兩個索引`queueFront`和`queueRear` [55] [56]。兩者的約定各實作不同，一種常見方式是： - 初始時令`queueFront = queueRear = 0`表示空佇列 [57]。 - 每次入隊：將元素放入`list[queueRear]`，然後`queueRear = (queueRear + 1) mod maxSize`前移一格 [58] [59]。- 每次出隊：先將`queueFront`也循環前移一格（跳過原本頭部），因此原頭元素視為刪除 [60] [61]。為了取得刪除元素值，可在移動前保存`list[(queueFront+1)%maxSize]`。 - 判斷空的條件是`queueFront == queueRear` [62]（初始即如此）。- 判斷滿的條件則通常設計為**預留一個空位**區分空滿：例如定義滿條件為**將要插入時**發現`(queueRear+1)%maxSize == queueFront` [57]。這樣陣列最多只能存`maxSize-1`個元素，始終留一格做為“佇列滿”的判斷標誌 [55] [63]。如果希望利用全部陣列空間，也可以用記錄元素數計算空滿，但會增加操作負擔。 - 另一做法：一些實作令`queueFront`指向對頭元素所在位置，而`queueRear`指向對尾元素所在位置**的下一個位置**。因此空時兩者相等，滿時插入前判斷`(queueRear+1)%maxSize == queueFront`。在這種約定下，對頭元素實際索引是`(queueFront+1)%maxSize` [64]。無論哪種策略，重點是要避免Front與Rear重合時混淆空或滿。 - **鏈結串列實作**：使用單向鏈結串列，保存一個指向頭節點的指標`front`和一個指向尾節點的指標`rear`。初始時兩者皆為`nullptr`表示空佇列。入隊時，在尾節點後插入新節點，並將`rear`指向新節點；若插入前佇列空，則同時更新`front`指向新節點。出隊時，將`front`指向原頭節點的下一節點，並刪除原頭節點；若刪除後佇列變空，記得將`rear`也設回`nullptr`。鏈結實作的優點是無固定容量限制，不需預留空位判斷滿，但缺點是需動態分配節點（稍有記憶體負擔）。

**佇列的典型應用**：- **廣度優先搜尋（BFS）**：在圖或樹的遍歷中，用佇列可以實現層序（廣度）搜尋，因為先被探索的節點會先將鄰居入隊，從而按探索順序先進先出地遍歷各層 [65]。- **緩衝區與工作排程**：佇列在操作系統中用來排程工作、緩衝周邊裝置資料（先到先服務），以及網路傳輸中的封包緩衝等 [66]。例如打印機工作列、作業系統的Ready Queue都是先到的任務先處理。 - **模擬排隊場景**：例如客戶服務中心的來人順序處理、流量控制等，都直接對應佇列行為。

### C++實作範例與程式碼說明

STL提供`<queue>`容器，默認以`deque`為底層，可直接使用：

```
#include <queue>
queue<string> line;
line.push("Alice");
line.push("Bob");
cout << line.front(); // 輸出 "Alice"
line.pop();       // 移除 "Alice"
```

自己實作一個循環陣列佇列的關鍵代碼可參考：

```
class Queue {
private:
```

```
    int maxSize;
    int queueFront, queueRear;
    int *list;
  public:
    Queue(int size=100) : maxSize(size), queueFront(0), queueRear(0) {
      list = new int[maxSize];
    }
    ~Queue() { delete[] list; }
    bool empty() const { return queueFront == queueRear; }
    bool full() const { return (queueRear + 1) % maxSize == queueFront; }
    void push(int item) {
      if (!full()) {
        list[queueRear] = item;
        queueRear = (queueRear + 1) % maxSize;
      } else {
        cerr << "Queue overflow\n";
      }
    }
    void pop() {
      if (!empty()) {
        queueFront = (queueFront + 1) % maxSize;
      } else {
        cerr << "Queue underflow\n";
      }
    }
    int front() const {
      if (empty()) throw runtime_error("Queue empty");
      return list[(queueFront + 1) % maxSize];
    }
    int size() const {
      return (maxSize + queueRear - queueFront) % maxSize;
    }
  };
```

在此實作中，使用了預留一空位法區分空滿：`full()`條件採用插入前判定下一位置會追上front。 57 67 。`push`將元素放入`queueRear`位置後移動`queueRear`；`pop`則移動`queueFront`跳過原頭元素。`front()`則取出`queueFront`的下一格內容（對應目前頭元素）。`size()`的計算為將`rear`與`front`之差調整至非負（注意取模）。需要強調的是，滿載狀態下，此實作最多只能有`maxSize-1`個有效元素。

### 時間複雜度分析

佇列的`enqueue`和`dequeue`操作本質上皆為指標（或索引）的移動和簡單賦值，因此時間複雜度為O(1)。`front()`和`back()`（若有）同樣O(1)。判空判滿也是O(1)。只有在特殊情況下，若使用動態陣列需要擴容（如Java的ArrayListQueue可能擴充底層陣列），擴容本身是O(n)，但攤銷後單次仍O(1)。鏈結串列實作中，`enqueue`需要配置節點O(1)，`dequeue`需要刪節點O(1)。因此佇列的操作效率相當穩定。

空間複雜度為O(n)。若用陣列，會配置一塊連續空間（可能有浪費一格的代價）；用鏈結串列則每個元素多一個指標。STL的`queue`（基於`deque`）則會動態調整，基本能視為O(n)。

### 可能的錯誤與除錯要點

- **未正確區分空滿**：循環陣列實作最容易出錯在判斷空滿條件。務必堅持一種約定並正確實現。如果預留一位法，記得在`size()`計算或其他地方考慮那個空位的存在。例如`size()`公式中我們用取模處理，就是避免直接`rear - front`在rear捲回小於front時出錯。
- **指標更新遺漏**：在鏈結串列實作中，入隊或出隊後漏更新`front`或`rear`。典型如佇列變空時忘了將`rear`重置`nullptr`；或空佇列首次插入時忘了設置`front`。這些都會導致後續操作錯誤。
- **內存泄漏**：鏈結串列節點出隊時如未釋放會內存洩漏。注意`pop()`要對刪除的節點`delete`。
- **調試佇列內容**：可以在每次操作後打印`front`和`rear`索引值（或指標所指值），以及佇列有效內容，來追蹤佇列狀態是否符合預期FIFO行為。對循環陣列而言，遍歷輸出時要從`front+1`開始數`size()`個元素（依模數算索引），檢查順序是否正確。

### 示意圖與補充說明

為幫助理解，想像一個佇列以陣列圓環排列：

```
位置:    0   1   2   ...   n-1
       [   ][   ][   ]... [   ]
           ^         ^
           front        rear
```

其中初始時`front`與`rear`同位置。每當有元素進佇列（enqueue），`rear`往右移一格放入元素，繞到陣列末端時再循環到開頭。每當有元素出佇列（dequeue），`front`往右移一格丟棄一個元素。`front`總是指向**上一個**對頭的位置（採上述約定時），而真正的對頭在`front`之後一格。 62 56 當`rear`追上`front`（差一格時），表示佇列已滿。

佇列的一大優點在於保持元素的順序，這對需要維持處理先後關係的場景非常重要。從實作角度看，環狀緩衝區（循環陣列）技巧不僅用於佇列，許多系統環形緩衝都是類似原理。透過充分測試基本操作和邊界情況（空佇列出隊、滿佇列入隊等），可以確保佇列實作的可靠性。

---

## 4. 二元樹（Binary Tree）

### 基本概念與術語定義

**二元樹**是一種樹狀階層式資料結構，其中每個節點最多只有兩個子節點（通常稱為左子節點和右子節點） 68 。形式上，二元樹要么是空的，要么由一個**根節點**和兩棵分別稱為其左子樹和右子樹的二元樹組成 68 。每個節點可看作包含**資訊欄位**（存放鍵值等資料）以及指向左右子節點的指標（或引用）。常用術語包括： - **根（root）**：樹的頂端節點，沒有父節點。 - **父節點（parent）**與**子節點（child）**：若節點A的子樹根節點是B，則稱A是B的父，B是A的子 69 。二元樹中每個節點最多有兩個子。 - **兄弟（sibling）**：同一父節點的兩子節點彼此為兄弟 70 。 - **葉節點（leaf）**：也稱終端節點，度數為0（無子）的節點 71 。 - **內部節點**：有至少一個子的節點（非葉）。 - **節點的度（degree）**：節點擁有子樹的數目，在二元樹中度只可能是0、1或2 71 。 - **層級（level）**：定義根節點在第0層（有些書以第1層），每往下一層孩子層級+1 72 。 - **深度（depth）或高度（height）**：通常指樹的最大層級。按照根為第0層定義，高度 = 樹中葉節點的最大層級值 73 。例如只有根一個節點的樹，高度為0（有的定義則為1，要注意定義差異 74 ）。 - **祖先（ancestor）與後裔（descendant）**：祖先指從根到該節點路徑上的所有節點，後裔則是以某節點為根的子樹中所有節點 75 。

## 二元樹的類型與性質

- **斷裂樹（Skewed Tree）**：極端情況，每個節點都只有左子或只有右子，使樹退化成鏈狀（高度接近節點數）。
- **滿二元樹（Full Binary Tree）**：常稱**嚴格二元樹**，每個內部節點恰有兩個子節點（0或2個，不存在只有1個子的節點） [76]。
- **完全二元樹（Complete Binary Tree）**與**幾乎完全二元樹（Almost Complete Binary Tree）**：對一棵有$n$個節點、深度為$d$的二元樹，若節點剛好填滿從根到深度$d$的全部位置（節點數$=2^{d+1}-1$），稱為**完全**；如果不是滿的，但節點的位置對應編號為0到$n-1$（按滿樹按層從左到右編號），則稱為**幾乎完全** [77]。簡言之，幾乎完全樹指除了最底層外，其餘皆滿，最底層節點集中在左側 [78]。一些文獻將“完全”定義為此情況，把滿的稱“滿”或“滿盈”，要注意名詞使用差異 [79]。
- **二元搜索樹（BST, Binary Search Tree）**：一種特殊二元樹，其中每個節點的左子樹所有節點鍵值均小於該節點鍵值，右子樹所有節點鍵值均大於該節點鍵值 [80]。BST允許以平均O(log n)時間進行搜尋、插入、刪除操作。需要注意處理重複值的約定（通常採不允許重複或將重複值放某一側一致處理）。
- **平衡二元樹**：為了避免普通BST在極端情況退化鏈狀，高度平衡的二元樹（如AVL樹、紅黑樹等）透過限制左右子樹高度差來保證操作效率。典型如AVL樹，每個節點左右子樹高度差不超過1，可證明AVL樹高度$O(\log n)$，因此操作$O(\log n)$ [81]（但插入刪除需旋轉維持平衡）。

## 二元樹的遍歷（Traversal）方法

遍歷指系統性地拜訪樹中所有節點的過程。二元樹有四種基本遍歷順序： - **先序（Preorder）**：訪問順序為 根 → 左子樹 → 右子樹 [82]。即先處理節點本身，再遞迴處理左子，再右子。 - **中序（Inorder）**：順序為 左子樹 → 根 → 右子樹 [83]。對BST中序遍歷將得到排序後的節點序列，常用於輸出有序數列。 - **後序（Postorder）**：順序為 左子樹 → 右子樹 → 根 [83]。常用於先處理子問題再處理根問題的場景，例如後序計算表達式樹值。 - **層序（Level-order）**：按樹的層級自上而下、每層從左至右順序訪問（這其實就是BFS）。需要輔助佇列實現。

例如，有如下二元樹：


其遍歷結果可能為： - Preorder: A, B, C, D, E, U, W, T, X, Y [83] - Inorder: D, C, E, B, A, U, W, T, X, Y [83] - Postorder: D, E, C, B, T, W, U, Y, X, A [83] - Level-order: A, B, X, C, U, Y, D, E, W, T。

遞迴實現前三種深度優先遍歷都較直接。然而在實務中也有**非遞迴遍歷**的需求，此時可使用**堆疊**輔助： - 先序非遞迴：用堆疊模擬，處理時先將右子壓棧再左子壓棧，這樣出棧時能先左後右訪問 [84]。 - 中序非遞迴：需要一路沿左子樹壓棧下去，回溯時訪問節點再轉向右子（典型算法利用指標一路push左子結點直到空，然後pop） [85]。 - 後序非遞迴：可藉由兩個堆疊或節點上標記法等實現，較為複雜，在此不展開。

### 二元樹的存儲與操作實作

**存儲表示**： - **鏈結表示**：每個節點結構包含資料及左右子節點指標：

```
struct TreeNode {
  Type info;
  TreeNode *left;
  TreeNode *right;
};
TreeNode *root;
```

使用時`root`指向根節點。不存在的子節點以`nullptr`標記。 - **陣列表示**（順序存儲）：可用一個陣列按照層序存放完全二元樹的節點。例如根在索引0，對於索引i，左子在`2*i+1`，右子在`2*i+2` 86 。這種方式適合**完全/幾乎完全二元樹**（如堆），節省指標空間且可以O(1)算出父子位置。但對於非完全樹會浪費大量空間（可能需要填充空洞），而且插入刪除不方便。

**基本操作實作**： - **插入節點**：依樹種不同而異。在一般無序二元樹，插入通常是指定位置插入；在BST，插入遵循大小比較，沿路徑尋找適當空位 87 88 。 - **搜尋**：在BST中可利用鍵值大小比較快速決定搜尋方向，每下降一層排除一半子樹，時間平均O(log n) 89 90 。遞迴或迭代實作皆可 91 89 。在一般無序二元樹，則只能用遍歷搜尋O(n)。 - **刪除節點**：BST中刪除較複雜，需分三種情況處理： 1. **葉節點**：直接刪除即可 92 。 2. **單子節點**（只有左或只有右子）：將此節點的唯一子提到原位置（讓父節點指向它的子） 93 。 3. **雙子節點**：尋找**中序後繼**（右子樹中最小節點）或**中序前驅**（左子樹中最大節點）取代之，然後刪除替代節點 94 。替代後仍保持BST性質。這部分較繁瑣，但是考試重點之一。實作時常使用遞迴。 - **遍歷輸出**：先前已述及，可用遞迴或非遞迴方法輸出節點值序列。

**範例**：建構二元樹。假設已知二元樹的中序與先序遍歷結果且各節點值唯一，可以遞迴構造原樹：

```
TreeNode* buildTree(vector<int>& inorder, int inL, int inR,
                    vector<int>& preorder, int preL) {
  if (inL > inR) return nullptr;
  int rootVal = preorder[preL];
  TreeNode* root = new TreeNode(rootVal);
  // 在中序序列找到根的位置
  int mid = find(inorder.begin()+inL, inorder.begin()+inR, rootVal) - inorder.begin();
  int leftSize = mid - inL;
  root->left = buildTree(inorder, inL, mid-1, preorder, preL+1);
  root->right = buildTree(inorder, mid+1, inR, preorder, preL+1+leftSize);
  return root;
}
```

這段程式以先序第一个元素為根，在中序序列找到根位置以劃分左右子序列，然後遞迴構建左右子樹。它運用了「先序確定根，中序劃分子樹」的原理，是典型構造問題。

### 樹的其他議題與應用

- **哈夫曼編碼（Huffman Coding）**：利用二元樹實現可變長度編碼的經典應用 95 。給定字元集和頻率，哈夫曼演算法每次選出權重最小的兩棵樹合併成新樹 96 97 。最終形成的哈夫曼樹的葉節點代表符號，從根到葉的0/1路徑即為該符號編碼 98 。哈夫曼樹構造使用**最小優先佇列（或小頂堆）**維護森林

中最小兩棵權值樹的選取 [99] 。演算法時間複雜度O(n log n)（每次從優先佇列取出最小兩項O(log n)，共n-1次合併）。哈夫曼編碼是壓縮領域的重要實例，展示了二元樹在編碼/譯碼中的作用：編碼時根據字元路徑拼接碼字，解碼時讀比特走樹直到葉節點得出符號 [98] 。

- **堆（Heap）**：堆其實是滿足特殊條件的**幾乎完全二元樹**。例如**最大堆（max-heap）**要求每個節點值不小於其子節點值 [100] ；**最小堆**則每節點值不大於其子值 [101] 。由於完全性質，堆通常用陣列存儲，以節省指標空間 [102] 。基本操作包括：
- **插入**：將新元素放在陣列尾（完全樹的下一空位），再**上濾**（percolate up）逐步與父節點比較交換，直到滿足堆序性。
- **取出堆頂（例如MaxHeap取最大值）**：移除根元素後，用最後一個元素填補根位置，再**下濾**（percolate down）將此元素下推與較大的子交換直到堆序恢復 [103] 。 這兩個操作時間皆為O(log n)，因為樹高度約$\log n$ [104] 。堆的重要應用是**優先佇列**，可在O(log n)時間插入和取出優先權最高的元素。
- **堆排序（Heapsort）**：利用堆特性進行排序的演算法。簡述：
- 將無序陣列建成最大堆 [105] 。可以從最後一個非葉節點開始下濾的方法線性時間完成建堆 [106] 。
- 反覆將堆頂（最大值）取出放到陣列末尾，並縮小堆的有效範圍，然後對新的堆頂下濾以恢復堆序 [107] [108] 。如此每次選出當前最大者，最終陣列從後往前即為升序。 堆排序最壞和平均時間複雜度都是$O(n \log n)$，且原地排序只需常數額外空間，是一種效率不錯的比較排序 [106] [109] 。其缺點是排序過程不穩定（不同順序的相等元素排序後相對次序可能改變）。
- **樹的等價與複製**：判斷兩棵二元樹是否結構相同且對應節點值相等，可用遞迴比較根以及左右子樹是否分別相等 [110] [111] 。複製一棵樹則遞迴創建新節點並拷貝左右子樹 [112] [113] 。
- **計算樹的性質**：如節點總數、高度、葉節點數等，都可設計遞迴函式實現。更有一些有趣問題如：檢查**完全性**（幾乎完全）的函式 `isAlmostComplete(tree)` ，可利用**層序遍歷**檢查是否有節點在出現空孩子後還有非空節點；或者計算**節點最小值**的函式 `getMin(tree)` ，可用遞迴或遍歷找最小 [114] 。
- **特殊樹種**：資料結構課程還可能涉及**AVL樹**（自平衡BST），**B樹/B+樹**（資料庫應用的多叉平衡樹），**紅黑樹**（平衡BST實作STL `set/map` 基礎）等，這些雖原理不同但核心目的都是保持操作效率在對數時間。

### 常見錯誤與除錯要點

- **遞迴邊界條件**：撰寫遞迴處理樹的函式時，別忘了處理空指標 `nullptr` （空樹或到達葉子下方）作為遞迴結束條件。例如比較兩樹相等函式中，兩節點同時為 `nullptr` 算相等、一方空一方不空則不等 [110] [111] 。
- **指標誤用**：在連結子節點時，若不小心將兩個節點指向同一子節點或遺漏連結，會造成結構錯誤甚至循環。構建樹時應小心地逐個設定 `left` 和 `right` 。特別是在刪除節點操作時，需謹慎調整子樹連結，不要遺留“野指標”引用已刪除節點。
- **平衡調整**：對於需要維持平衡的樹（如AVL），插入和刪除後必須正確執行旋轉。學生常見錯誤是在實現複雜旋轉時判斷條件或子樹引用搞錯，需對照教材圖解一步步推演檢查。
- **樹的遍歷順序**：在套用遍歷結果構造或判斷問題時，搞清遍歷定義非常重要。例如前序和後序不能唯一確定一棵樹（可能有多種樹產生相同前後序組合）。在推導題目時，可透過畫小範例樹來驗證理解。
- **測試**：使用各種特殊樹結構來測試，例如極端不平衡樹、只有左或只有右的鏈、完全二元樹、只有一個節點或空樹等，確保程式對所有情況都正確。

---

## 5. 補充主題

### 哈夫曼編碼

**哈夫曼編碼**是資料壓縮領域的重要演算法，用於根據符號出現頻率產生最優的前綴碼（沒有碼是另一個碼的前綴） [115] [98] 。其核心在於構造一棵**哈夫曼樹**：權重（頻率）較小的節點會離根較遠，確保常用符號擁有較短的編碼長度。演算法流程： 1. 將每個符號建立為只有一個葉節點的樹，權重為該符號頻率 [116] 。 2. 將這些樹放入

一個**最小優先佇列**（或使用最小堆）根據權重排序[97]。 3. 重複以下步驟直到佇列中只剩一棵樹： - 取出權重最小的兩棵樹`t1`和`t2`[99]。 - 建立一個新樹`t`，設定`t1`和`t2`為其左、右子樹，新根的權重為兩子權重和。 - 將新樹`t`插回佇列。 4. 剩下的最後一棵樹即為哈夫曼樹[99]。從根到葉每經過左枝記0，右枝記1，即可為每個符號生成對應的二進位碼。

哈夫曼樹是一棵**最優二元樹**，其帶權路徑長度（WPL）最小。哈夫曼編碼能保證無損壓縮下的最短總編碼長度。譬如，有符號A,B,C,D權重分別為5,7,10,16（假設），使用固定長度2位編碼需要2*4=8位表示一個符號，但用哈夫曼碼可能得到不同位數的編碼（如A:110, B:111, C:10, D:0），使得整體訊息編碼長度下降。編碼時依次串接各符號碼字成比特流，解碼時從比特流起頭開始在哈夫曼樹遍歷：讀0走左，讀1走右，直到走到葉節點就輸出對應符號，再回到根節點繼續[98]。

需要注意的是，哈夫曼編碼產生的碼長為變長且不含前綴關係，這保證了解碼時不會混淆。哈夫曼演算法的效率主要取決於每次挑選最小兩項，若用最小堆實現每次取出插入都是O(log n)，n個符號需要合併n-1次，因此總計O(n log n)。作為補充，高度相近的哈夫曼樹其左右子樹權重相差不會太懸殊，但一般不是平衡樹。

## 堆積（Heap）與優先佇列

**堆積**是一種特殊的**幾乎完全**二元樹，同時滿足堆積順序特性： - **最大堆**：每個節點的值 >= 其孩子節點的值[100]。因此根節點保存全局最大值。 - **最小堆**：每個節點的值 <= 其孩子節點的值[101]。因此根節點是全局最小值。

由於樹高度約$\log n$且結構接近完全，可用一維陣列高效存儲：節點i的左子在`2*i+1`，右子在`2*i+2`[86]；父節點在`floor((i-1)/2)`[117]。常見操作： - **取堆頂**：O(1)直接返回陣列[0]（根）。 - **插入**：將元素放在陣列尾，然後上濾：與父比較，若違反堆序則交換，重複直到父子順序正確[105]。最壞需上濾到根，高度約$\log n$，所以O(log n)。 - **刪堆頂**（例如取出最大值）：將最後一個元素移到根（覆蓋原根），陣列長度減一，然後下濾：從根開始與較大的子（對max-heap）比較，若子大於目前節點則交換，繼續向下直到沒有子或不需交換[118]。這同樣至多下降$\log n$層，O(log n)。 - **建堆**：可以將n個元素直接組成陣列，從最後一個非葉節點開始倒序對每個節點下濾調整。最後一個非葉節點索引大約是$n/2-1$，向前遍歷每個節點各下濾一次。每個節點下濾成本與其高度相關，高度h的節點最多下濾h步。高度為0的有約n/2個，1的有n/4個...計算總耗時$ \sum_{h=0}^{\lfloor \log n \rfloor} \text{(數量)} * O(h)$，結果為O(n)[106][119]。因此可線性時間建堆，而非逐一插入的O(n log n)。

**優先佇列**便是以堆為底層實現的一種資料結構。它支援插入元素和提取最高優先權元素（相當於最大堆取最大值）的操作，均為$O(\log n)$. C++ STL的`priority_queue`預設即為最大堆。應用遍及演算法（如Dijkstra最短路、Prim最小生成樹等需要反覆取得當前最小權重邊/點）、資源調度（如CPU任務按優先等級執行）等。

**注意**：堆積並非二元搜索樹，不支援一般性的搜尋某個鍵（只能線性掃描O(n)或取頂O(1)）。它的用途側重在重複**取極值**的場景。

## 排序演算法總覽

課程涵蓋了一些重要的排序方法及其性能： - **冒泡排序（Bubble Sort）**：重複地兩兩比較相鄰元素，將較大（或較小）者逐步冒泡到序列一端。每一輪會將一個最大值放到尾端，需要n-1輪完成排序。最壞和平均時間複雜度$O(n^2)$，幾乎已不在實務使用，但易於理解。冒泡排序是**穩定排序**（相等元素相對次序保持）。 - **插入排序（Insertion Sort）**：模擬抓撲克牌插牌的動作，維護一段已排序序列，依次將新元素插入適當位置。對於接近有序的輸入，插入排序表現很好，最壞$O(n^2)$但最佳情形$O(n)$（已排序時只需一遍掃描）。插入排序通常也是穩定的。適用於小規模資料或幾乎有序資料。教材可能要求手動實作其模板版本[120][121]。 - **選擇排序（Selection Sort）**：每輪從未排序部分選出最小（或最大）元素，放到前端已排序部分後。進行n-1輪。最壞和平均也是$O(n^2)$，而且因無法提早結束，通常比插入排序慢。選擇排序是不穩定的（因為交換可能改變相等值順序）。 - **快速排序（Quick Sort）**：分治法排序的代表。隨機選取一個**樞紐（pivot）**，將序列分區為

左側都小於pivot、右側都大於pivot，然後遞迴排序兩側。平均時間$O(n \log n)$，且常為實際排序中最快者之一。但最壞情況（例如輸入已排好序且每次選到的pivot為最大或最小）會退化到$O(n^2)$，通常透過隨機選pivot或三點中值法降低最壞情況概率。快速排序通常**不穩定**。在課程lab中，可能使用了C函式庫的`qsort()`作比較[122][123]。 - **合併排序（Merge Sort）**：將數列平分兩半，各自排序後再線性合併兩個有序序列。遞迴深度$\log n$，每層合併成本$O(n)$，總計$O(n \log n)$穩定時間；但空間複雜度需要$O(n)$輔助陣列。不像快排，合併排序最壞也是$O(n \log n)$，因此在有穩定性或最壞性能要求時很適用。 - **堆排序（Heap Sort）**：前述利用堆取代樞紐分區的選擇法。建堆O(n)，每次取最大O(log n)共n次，總$O(n \log n)$[106]。因不穩定性與cache局部性差（跳著訪問陣列），實際表現常略遜於快排，但勝在空間原地且有保證上界。 - **計數排序、桶排序、基數排序**：如果數據特性允許（如整數範圍有限或可拆位處理），這些**線性時間排序**可能會在課程提及，但它們不是基於比較的排序法，不在本次複習重點。

**排序演算法比較**： - 時間複雜度方面，$O(n^2)$的簡單排序（冒泡、插入、選擇）僅適用於小n；$O(n \log n)$的高級排序（快排、合併、堆）能處理大資料。特別提及，在lab實驗中可能會讓學生測試不同排序在不同n下的執行時間[124][125]。 - 穩定性方面，如需要保持相等元素順序（例如按照某鍵排序後再按另一鍵排序需要前一排序穩定），會選擇穩定算法（插入、合併）或對快排進行改良。 - 原地性方面，插入、選擇、快排、堆排都是就地排序（in-place，不需要額外大量記憶體），合併排序則需要額外陣列。 - 最壞情況保障：合併和堆有保證，快排沒有但平均更優。實作中C++的`std::sort`使用了一種混合算法（Introspective sort），一般情況用快速排序，當遞迴深度過深時改用堆排序保證$O(n \log n)$最壞[126][123]。

## 綜合除錯要點

- **邊界條件**：排序演算法要小心處理陣列索引邊界和子序列區間。例如快速排序遞迴時區間劃分、合併排序merge時兩段結束條件，都易出錯。建議畫圖或手模擬小數據來驗證正確性。
- **原地或輔助數組**：實現時，合併排序若寫成原地版較困難，一般用輔助陣列；而快排、堆排要確保交換時正確操作全域陣列，不要錯用局部。
- **函式介面**：使用`qsort`或`std::sort`需要提供比較函式或函式對象，容易發生比較函式寫錯（例如沒有按照需求完全比較所有鍵值就返回0導致認定相等）。lab可能要求實作自定義比較函式，例如比較結構體的某字段[127][128]。
- **效率檢查**：對於較慢的排序，當輸入規模稍大時（例如幾萬以上）性能會明顯下降。可通過記錄執行時間來對比不同演算法效率[124][125]。這能加深對複雜度概念的理解，也幫助找到實作中潛在低效之處（如不必要的重複內圈計算等）。

---

本複習筆記以鏈結串列、堆疊、佇列、二元樹等核心資料結構為主軸，輔以哈夫曼編碼、堆和排序算法等延伸議題，涵蓋了課程檔案中的重點內容。在掌握每種結構的定義、操作方法和實作細節的同時，務必理解其時間空間性能和典型應用場景。透過對比不同結構的優劣、模擬演練程式碼和多畫圖輔助思考，相信能夠加深對資料結構概念的理解，為期中考做好準備。

---

[1][2][3][4][5][6][7][8][9][10][11][12][13][14][15][16][17][18][19][20][21][22][23][24][25][26][27][28] 05 - linked list.pdf

file://file_00000000090c720898955be506a95982

[29][30][31][32][33][34][39][40][41][42][43][44][45][46][47][48][49][50][51][52][53][54][55][56][57][58][59][60][61][62][63][64][65][66][67] 06 - Stack and Queue.pdf

file://file_00000000f5f0720890e5c0e14c690e8b

[35][36][37][38] tutorial_9.cpp

file://file-256vFXnPyM1XZagLQACQWN

68 69 70 71 72 73 74 75 76 77 78 79 80 81 82 83 84 85 86 87 88 89 90 91 92 93 94 95 96
97 98 99 100 101 102 103 104 105 106 107 108 109 110 111 112 113 115 116 117 118 119 07 - binary tree.pdf

file://file_0000000033c072089a8ffa1ad8360762

114 tut_10.cpp

file://file-ABsir9NMMveBSbX9RFQnMe

120 121 122 123 124 125 126 127 128 tut_06.cpp

file://file-8SGdQ4NbXgkNZuo7bbUuj5