

EECS 281: Data Structures and Algorithms

Summary Part 1

University of Michigan at Ann Arbor

Last Edit Date: 06/26/2023

Disclaimer and Term of Use:

1. We do not guarantee the accuracy and completeness of the summary content. Some of the course material may not be included, and some of the content in the summary may not be correct. You should use this file properly and legally. We are not responsible for any results from using this file.
2. This personal note is adapted (derived) from the slides and codes from *Professor Marcus Darden and David Paoletti*. Please [contact us](#) to delete this file if you think your rights have been violated.
3. This work is licensed under a [Creative Commons Attribution 4.0 International License](#).

1. Stack (Last in first out - LIFO)			
Method	Description	Array	Linked List
push(obj)	Add object to top of the stack	$O(1)^{1)}$	$O(1)$
pop()	Remove top element	$O(1)$	$O(1)$
obj &top()	Return reference of top element	$O(1)$	$O(1)$
size()	Number of elements in stack	$O(1)$	$O(1)^{2)}$
empty()	Check if stack has no element	$O(1)$	$O(1)$

Examples: web browser's "back" feature; function calls in C++, ...

STL Tip: `std::stack<>`, use `#include <stack>`. The default underlying container is `std::deque<>`, some optional underlying containers are `std::list<>` (doubly-linked) and `std::vector<>`.

2. Queue (First in first out - FIFO)			
Method	Description	Array	Linked List
push(obj)	Add object to back of the queue	$O(1)^{1)}$	$O(1)$
pop()	Remove front element	$O(1)$	$O(1)$
obj &front()	Return reference of front element	$O(1)$	$O(1)$
size()	Number of elements in queue	$O(1)$	$O(1)^{2)}$
empty()	Check if queue has no element	$O(1)$	$O(1)$

Examples: waiting line for lunch, adding songs to a playlist, ...

STL Tip: `std::queue<>`, use `#include <queue>`. The default underlying container is `std::deque<>`, an optional underlying container is `std::list<>` (doubly-linked).

3. Deque			
Method	Description	Array	Linked List
push_front()	Add object to front of the deque	$O(1)^{1)}$	$O(1)$
pop_front()	Remove front element	$O(1)$	$O(1)$
front()	Return front element	$O(1)$	$O(1)$
push_back()	Add object to back of the deque	$O(1)^{1)}$	$O(1)$
pop_back()	Remove back element	$O(1)$	$O(1)$
back()	Return back element	$O(1)$	$O(1)$
size()	Number of elements in queue	$O(1)$	$O(1)^{2)}$
empty()	Check if deque has no element	$O(1)$	$O(1)$

We can consider deque as a combination of stack and queue. Can traverse using iterators.

STL Tip: `std::deque<>`, use `#include <deque>`. It can have stack / queue-like behavior at both ends. Random access with `[]` or `.at()`.

Note: 1) when pushing with a vector, the time complexity is $O(n)$, because it averages out to constant with many pushes (amortized constant); **2)** when getting the size of a linked list, our time complexity is $O(1)$ when we have a tracked size, otherwise, we need to go over it.

4. Priority Queue (customizable)	
Method	Description
push(obj)	Add object to the priority queue
pop()	Remove highest priority element
const obj &top()	Return a reference to highest priority element
size()	Number of elements in priority queue
empty()	Check if priority queue has no element

Examples: Hospital queue for arriving patients, load balancing on servers.

STL Tip: `std::priority_queue<>` maintain a heap in any random access container such as `#include <queue>`.

Common `std::priority_queue<>` declarations:

- "Max" PQ using `std::less<>`: `std::priority_queue<T> myPQ;`

- PQ using a custom comparator type, **COMPARATOR**:

`std::priority_queue<T, vector<T>, COMPARATOR> myPQ;`

Time Complexity with different containers

Container	Insert	Remove
Unordered sequence container	$O(1)$	$O(n)$
Sorted sequence container	$O(n)$	$O(1)$
Heap	$O(\log n)$	$O(\log n)$
Array of linked lists (for priorities of small integers)	$O(1)$	$O(1)$

Use `#include <algorithm>`, `std::make_heap()`, `std::push_heap()`, and

`std::pop_heap()` to manually implement priority queue.

Note: By default `std::priority_queue<>` uses `std::less<>` to determine relative priority of two elements. A "default PQ" is a "max PQ", where the **largest element has highest priority**. If a "min PQ" is desired, customize with `std::greater<>`, so the **smallest element has highest priority**. If the PQ will hold elements that cannot be compared with `std::less<>` or `std::greater<>`, customize with custom comparator (function object).

5. Complexity Analysis

1. Metrics of Algorithm Complexity

- **Best case:** Least number of executions required, given ideal input.
- **Average case:** Average number of executions required, given any input.
- **Worst case:** Most number of executions required, given hard input.

2. Factors Affect Runtime

University of Michigan – EECS 281 Summary Part 1

The algorithm; implementation details; CPU speed / memory speed; compiler (options used); other programs running in parallel; input size. Rate of growth is independent of most factors such as CPU speed, compiler, ...

3. Measuring Using Input Size

1) Number of bits: `bool` – 1 byte; `char` – 1 byte; `int` – 4 bytes; `float` – 4 bytes; `double` – 8 bytes; `size_t` – 4 (32-bit) or 8 (64-bit) bytes (unsigned). **2) Number of items 3) Notation:** n – input size; $f(n)$ maximum number of steps taken by an algorithm when input has size n , $O(f(n))$ complexity class of $f(n)$.

6. Counting Steps

- **One step:** primitive operations - variable assignment, arithmetic operation, comparison, array index, pointer reference, function call (not counting the data), function run (not counting the date). Independent on input.

- **for loop count:** for (initialization; test; update)

The initialization is performed once (1); the test is performed every time the body of the loop runs, plus once for when the loop end ($n + 1$); the update is performed every time the body of the loop runs (n).

<pre>1 int function(int n) { 2 int sum = 0; 3 for (int i = 0; i < n; i++) 4 sum += 1; 5 return sum; 6 }</pre>	<pre>1 int function(int n) { 2 int sum = 0; 3 for (int i = 0; i < n; i /= 2) 4 sum += 1; 5 return sum; 6 }</pre>
<pre>2 1 step 3 1 + n + 1 + n steps 4 1 step (runs n times total) 5 1 step 3n + 4 in total $O(n)$</pre>	<pre>2 1 step 3 1 + log n + 1 + log n steps 4 1 step (runs log n times total) 5 1 step 3log n + 4 in total $O(\log n)$</pre>

7. Big-O, Big-Θ, and Big-Ω

Big-O upper bound **Big-Θ** tight bound **Big-Ω** lower bound
 $f(n) = O(g(n))$ iff. $f(n) = \Theta(g(n))$ iff. there $f(n) = \Omega(g(n))$ iff.
 is an integer n_0 and a real number c such that for all $n \geq n_0$,
 $f(n) \leq c \cdot g(n)$.
 is an integer n_0 and real constants c_1, c_2 such that $c_1 \cdot g(n) \leq f(n) \leq c_2 \cdot g(n)$.
 there is an integer n_0 and a real number c such that for all $n \geq n_0$, $f(n) \geq c \cdot g(n)$.

Common Big-O: $O(1)$ – constant; $O(\log n)$ – logarithmic; $O(n)$ – constant;

$O(n \log n)$ – log-linear; $O(n^2)$ – quadratic; $O(n^3)$, $O(n^4)$, ... - polynomial; $O(c^n)$ – exponential; $O(n!)$ – factorial; $O(2^{2^n})$ – doubly exponential.

Note: If $\lim_{n \rightarrow \infty} \frac{f(n)}{g(n)} = d < \infty$, then $f(n) = O(g(n))$. Lower-order terms and coefficients of highest-order terms can be ignored: $3n^2 + n = O(n^2)$.

8. Amortized Complexity

It is a type of worst-case complexity. Considers the average cost of one operation over a sequence of operations. Key to understand vectors.

9. Recursion

1. Types of Recursion

- **Linear Recursive:** A function is linear recursive if it is recursive, but each invocation of the function makes **at most one recursive call** (each recursive case is a subproblem).
 - **Tail Recursive:** A function is tail recursive if it is linear recursive, and every recursive call is **the last thing that happens in the invocation** that makes the recursive call.

2. Solving Recurrences

Method 1: Substitution Method

$T(n) \in \begin{cases} \Theta(n^{\log_b a}) & \text{if } a > b^c \\ \Theta(n^c \log n) & \text{if } a = b^c \\ \Theta(n^c) & \text{if } a < b^c \end{cases}$

1) Write out $T(n)$, $T(n - 1)$, $T(n - 2)$; **2)** substitute $T(n - 1)$, $T(n - 2)$ into $T(n)$; **3)** look for a pattern; **4)** use a summation formula.

Method 2: Master Theorem

- Let $T(n)$ be a monotonically increasing function that satisfies $T(n) = aT(\frac{n}{b}) + f(n)$

and $T(1) = c_0$ or $T(0) = c_0$, where $a \geq 1$, $b > 1$. If $f(n) \in \Theta(n^k)$, then:

- For polylogarithmic functions, if $f(n) \in \Theta(n^{\log_b a} \log^k n)$ for some $k \geq 0$, then $T(n) \in \Theta(n^{\log_b a} \log^{k+1} n)$. This is the fourth condition.

Note: We cannot use the Master Theorem if **1) $T(n)$ is not monotonic** ($T(n) = \sin n$); **2) $f(n)$ is not a polynomial** (ex: $f(n) = 2^n$); **3) b cannot be expressed as a constant**; **4) the recursion does not involve division**; **5) $T(n) \neq aT(n/b) + f(n)$.**

10. Arrays and Containers

- Basics of Array in C / C++:** **1)** Allows random access in $O(1)$ time complexity; **2)** index numbering starts at 0; **3)** Size must be separately stored; **4)** no bounds checking.
- Range-based Loops - `for (int &i : arr)`:** Either passed by value or reference.
- Big-5:** Destructor, copy constructor, overloaded `operator=()`, copy constructor from right hand side value, and overloaded `operator=()` from right hand side value.
- Pointer Tips:** **1)** `new / new[], delete / delete[]`; **2)** assign `nullptr` after deletion.
- `const operator[]` accesses const data. Copy ctor - $O(n)$, insert - $O(1)$ or $O(n)$.

	Arrays	Linked Lists
Access	Random in $O(1)$ time Sequential in $O(1)$ time	Random in $O(n)$ time Sequential in $O(1)$ time
Insert Append	Inserts in $O(n)$ time Appends in $O(1)$ time (vector)	Inserts in $O(n)$ time ($O(1)$ if given location) Appends in $O(n)$ time ($O(1)$ if tail pointer)
Memory	Wastes memory if oversized Wastes time if repeatedly undersized	Allocates memory as needed Requires memory for pointers

11. Standard Template Library

1. Containers: `vector<>`, `deque<>`, `bit_vector`, `set<>`, `multi_set<>`, `map<>`, `multi_map<>`, `list<>`, `array<>`, `list<>` (doubly), `slist<>`, `forward_list<>`.

2. Iterator: `begin()`, `end()`, `cbegin()`, `cend()`, `rbegin()`, `rend()`. "begin" is inclusive, "end" is exclusive. Iterators are faster than indices. No need random access.

12. Heap and Heapsort

1. Tree: **Root** – the "topmost" node in the tree; **parent** – immediate predecessor of a node; **child** – node where current node is parent; **ancestor** - parent of a parent (closer to root); **descendant** – a node with children; **leaf node** - a node without children.

2. Binary Tree: every node has 2 or fewer children; **complete binary tree:** every level, except possibly the last, is completely filled, and all nodes are as far left as possible.

3. Heaps: data structure that gives easy access to the most extreme element ("max heap" is heap with largest element being the "most extreme"; "min heap" vice versa).

4. Priority Queue: unordered array PQ - $O(1)$ insertion, $O(n)$ inspection top, $O(n) / O(1)$ remove top; sorted array PQ - $O(n)$ insertion, $O(1)$ inspection top, $O(1)$ remove top; heap - $O(\log n)$ insertion with `fixup()`, $O(1)$ inspection top, $O(\log n)$ remove top with `fixDown()`.

5. Heapsort: **1)** Take the given n elements, convert into a heap (heapify takes $O(n)$ time); **2)** Remove elements one at a time, filling original array from back to front (swap top with last unsorted $O(1)$, `fixDown()` to bottom: each $O(\log n)$ time) $O(n \log n)$ total.

13. Sorts

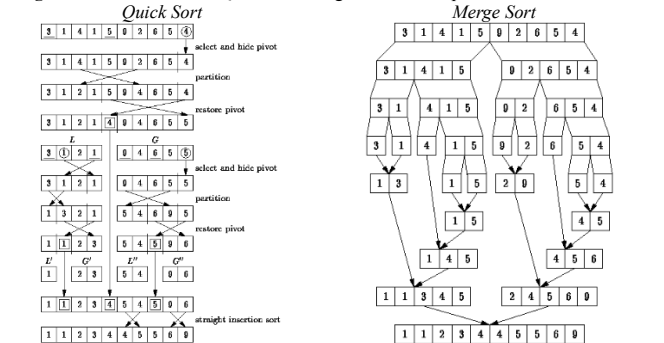
1. Elementary Sorts: bubble sort, selection sort, insertion sort, and counting sort.

Bubble Sort	Selection Sort	Insertion Sort	Counting Sort

Note: Only useable with small, limited keys.

Note: The above are non-adaptive sorts, sequence of operation is independent. Adaptive sort performs different sequences of operations depending on outcomes of comparisons.

2. High Performance Sorts: Quick sort, merge sort, and heap sort



3. Complexity Analysis

Sort	Best	Average	Worst	Memory	Stable
Bubble	$O(n^2)$	$O(n^2)$	$O(n^2)$	$O(1)$	Yes
Selection	$O(n^2)$	$O(n^2)$	$O(n^2)$	$O(1)$	No
Insertion	$O(n^2)$	$O(n^2)$	$O(n^2)$	$O(1)$	Yes
Quick	$O(n \log n)$	$O(n \log n)$	$O(n^2)$	$O(\log n)$ extra	No
Merge	$O(n \log n)$	$O(n \log n)$	$O(n \log n)$	$O(n)$	No

14. Iterator Summary Table

	Input	Output	Forward	Bidirectional	Random
Supports dereference (*) and read		✓		✓	✓
Supports dereference (*) and write			✓	✓	✓
Supports forward movement (++)	✓	✓	✓	✓	✓
Supports backward movement (--)				✓	✓
Supports multiple passes			✓	✓	✓
Supports == and !=	✓		✓	✓	✓
Supports pointer arithmetic (+, -, etc.)					✓
Supports pointer comparison (<, >, etc.)					✓