Some Datastructure and Algorithm Summary
The time complexity of the algorithm

Kim JaeHwan

---

1 Simple Meaning of main properties of C++

- Object-oriented
  the benefits of OOP?

- Generic programming
- Abstract data types
- Information hiding
- encapsulation

2 About the time complexity

2.1 Definition and time complexity

Algorithm : a step-by-step procedure for solving a problem in a finite amount of time.

How to comopare two algorithms?
Efficiency - Running time (time complexity), Space requirements (space complexity)

There are two method to check time complexity (1. Empirical studies, 2. Theoretical analysis). First method is programming and testing, but it is not exact. It can be changed by the environment such as hardware and software. So, we need to use theoretical analysis. Theoretical analysis is a high-level description of the algorithm instead of an implementation. No need to be so exact. There are three ways to compare algorithms. (1. Best case, 2. Average case, 3. Worst case)

1. Best case(lower bound) : **Big-Omega notation**
   $T(n)$ is $\Omega(f(n))$ if there exist a constant $c > 0$ and an integer constant $n_0 \geq 1$ such that $T(n) \geq cf(n)$ for all $n \geq n_0$.
2. Average case : **Big-Theta notation**
   $T(n)$ is $\Theta(f(n))$ if $T(n)$ is $O(f(n))$ and $\Omega(f(n))$.
3. Worst case(upper bound) : **Big-Oh notation**
   An algorithm is $O(f(n))$ if there exist a constant $c > 0$ and an integer constant $n_0 \geq 1$ such that $T(n) \leq cf(n)$ for all $n \geq n_0$. Then we write $T(n) \in O(f(n))$, or $T(n) = O(f(n))$.

   This case is easier to analyze.

2.2 Properties of Big-Oh

Addition rule, Product Rule, and others.
There are typical growth rates:

$\Theta(1)$	constant	$\Theta(n)$	linear
$\Theta(\log n)$	logarithmic	$\Theta(n \log n)$	log linear
$\Theta(n^2)$	quadratic	$\Theta(n^3)$	cubic
$\Theta(2^n)$	exponential	$\Theta(n!)$	factorial

limitation of analysis

- Not account for constant factors, but constant factor ay dominate.
- Not account for different memory access times at different levels of memory hierarchy.
- Programs that do more computation may take less time than those that do less computation.

For example, 1000n vs n^2, when interested only in $n < 1000$. For example, Cache memory << MM << HDD.

2.3 Relative of Big-Oh

1. Little-oh
2. little-omega

3 List, Stack, Queue

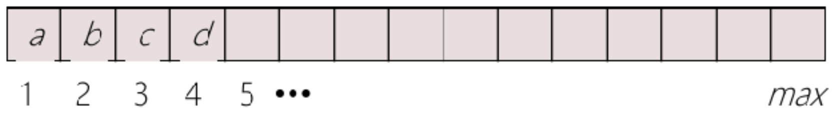
1. List

Definition

A finite, **ordered** collection of elements. n. length (size) of the List

List Operations: Insert(x, p, L), Delete(p, L), Next(p, L), Previous(p, L), Locate(p, L), Retrieve(p, L), MakeNull(L), First(L)

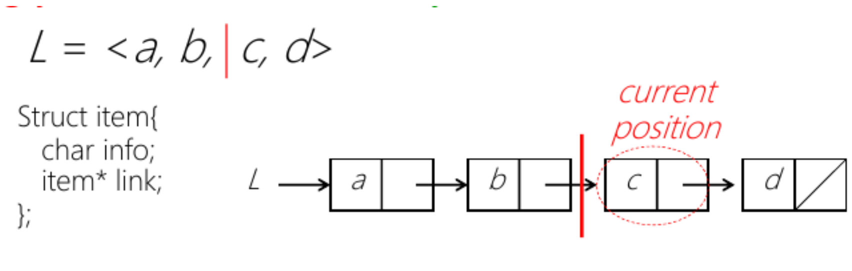
Array-based list



This, size n = 4 If execute Insert(x, 2, L), then b, c, d will be moved to the right side, then arr[index = 2] will be x.

TIME COMPLEXITY : **Insert(x, p, L)** : $O(n)$ **Delete(p, L)** : $O(n)$ **Locate(p, L)** : $O(1)$

Pointer-based list



```
Struct item{
  char info;
  item* link;
};
```

- Singly-linked(one-way) : each node has a pointer to the next node, and value.

1. P direcctly points to the current element. Difficulty for insert : hard to access to the preceding node of the current one, but we have to change the link of the preceding node. -¿ we need a pointer to the preceding node.
2. **One-step ahead convention** : P points to the previous element of the current position. P is the position of the preceding node.
   When executing Insert(x, p, L) and the list is empty or the left node is empty, we can't use. Solution : add a dummy node at the beginning of the list.

If you know the location(pointer) of the preceding node, then you can insert the new node in $O(1)$ time. But if you don't know, the time complexity is $O(n)$.

- Doubly-linked(two-way) : each node has a pointer to the next node, and the previous node, and value. **Q**. what is the difference between singly-linked and doubly-linked? **Solution**: In singly-linked, we can't access to the preceding node of the current one, but in doubly-linked, we can access to the preceding node of the current one. So we can get the previous node of the current one in $O(1)$ time.

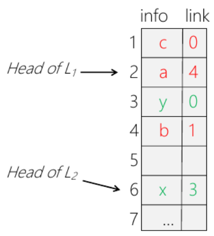
The Comparison of singly-linked and doubly-linked is below.

	Singly-linked	**Doubly-linked**
Insert(x, p, L)	$O(n)$	$O(n)$
Delete(p, L)	$O(n)$	$O(n)$
Locate(p, L)	$O(n)$	$O(n)$

It is similar to the singly-linked list, but we can access to the preceding node of the current one. So we can get the previous node of the current one in $O(1)$ time.

Cursor-based list

Cursor : simulated pointer. Interger index indicating positions in array to simulate pointers.
TIME COMPLEXITY : insert(x, p, L) : $O(1)$, delete(p, L) : $O(1)$, retrieve(p, L) : $O(n)$



2. Stack

Definition

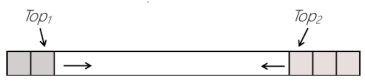
All insertion & deletions take place at one end(Top). LIFO(Last In First Out) structure. You can implement stacks using any type of list implementation(pointer, array, cursor, ...). Stack Operations: Push(x, S), Pop(S), Top(S), MakeNull(S), IsEmpty(S)

Array-based Stack

How to implement TOP? (in terms of cost of pop/push)
Position k(when k elements in stack): $O(1)$
(Fixed) Position 1: $O(n)$

Linked stack

Very much similar to the pointer-based list implementation



3. Queue

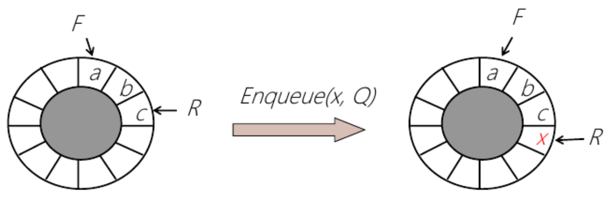
Definition

FIFO(First In First Out) structure.
Similar to top in the stack, here we have front and rear Queue Operations: Enqueue(x, Q), Dequeue(Q), Front(Q), MakeNull(Q), IsEmpty(Q)

Array-based Queue

- Simple array implementation
  "drifting queue" problem. we can solve in an inefficient way. Enqueue & dequeue : $O(1)$ & $O(n)$ or vice versa.

- **Circular Queue**



  Modulus / Modulo operator : Methematically connect the last element to the first element via modulo operator.
  Then, we have a problem. How to recognize empty queue or full queue?
  Sol 1. Explicit count variable such as cnt.
  Sol 2. Boolearn variable 'isEmpty'

Linked Queue

- without dummy header, Two pointers : (F, R) Empty(Q) is true if F = R = NULL.
  Enqueue(x, Q) also can, but Problem: Can we use the same code when inserting into an empty queue?

- with dummy header, Empty(Q) is true if F = R = header.

Comparison with header and without header. Speed? Space utilization? Code conciseness? - insertion into an empty queue, deleteion when queue has only one.

Others

There are other types of queues: queue, circular queue, priority queue, double-ended queue(deque), ...

4 Code Testing

- The ability to test your own code is integral to an understanding of data structures.
  - Differentiating between requirements and design decisions you made.
  - Coming up with test cases is one of the best ways to understand data structures more deeply.
    - What cases will cause certain implementations to slow down?
    - How long do I expect certain operations to take?
    - What edge cases are there in the definition?
    - Where else might I find bugs?
- In the real world, coding projects don't come with their own tests.
  - You have to write your own.
- Learning to test your own code is integral to maturing as a computer scientist.
- Types of Tests
  - Black Box
    - Behavior only - requirements
    - From an outside point of view
    - Does your code uphold its contracts with its users?
    - Performance/efficiency
  - White Box
    - Includes an understanding of the implementation
    - Written by the author as they develop their code
    - Break apart requirements into smaller steps
    - "unit tests" break implementation into single assertions
- What to Test?
  - Expected behavior
    - The main use case scenario
    - Does your code do what it should given friendly conditions?
  - Forbidden Input
    - What are all the ways the user can mess up?
  - Empty/Null
    - Protect yourself!
    - How do things get started?
    - 0, -1, null, empty collections
  - Boundary/Edge Cases
    - First items
    - Last item
    - Full collections (resizing)
  - Scale
    - Is there a difference between 10, 100, 1000, 10000 items?

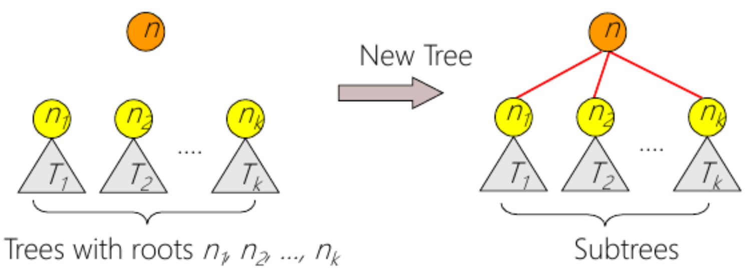
5 Tree

Definition

A tree is collection of nodes and edges. with one node distinguished as a root, along with parenthood relation, one edge connects two nodes. Each node in the tree can be connected to many children, but must be connected to exactly one parent, except for the root node. No cycles or loops

Recursive Definition

Starts with a single node, then we can build a new tree by making other tress as subtrees of the single node.



Some keywords : Parent/child, Ancestor/descendant, Siblings, Leaf, Depth, Height, Level.

Binary Tree : Every node has at most two children, Each child is designated as left child or a right child. (left child and right child are distict.)
Proper Tree : Every node has 0 or 2 children.
Full Tree : If it has a maximum number of nodes at each level. A full binary tree of height h has $2^{h+1} - 1$ nodes.

Node numbering :

1. Zero-based numbering:

Array	0	1	2	3	4	5	6	7	...

   Node 0 is the root, and the left child of node i is 2i+1, and the right child of node i is 2i+2. Parent node is $\lfloor \frac{i-1}{2} \rfloor$.

2. One-based numbering:

Array	1	2	3	4	5	6	7	8	...

   Node 1 is the root, and the left child of node i is 2i, and the right child of node i is 2i+1. Parent node is $\lfloor \frac{i}{2} \rfloor$.

Complete Binary Tree : Relaxed definition of a full binary tree. A binary tree of height h is complete, if all levels possibly except h are completely full, and level h is filled from left to right.

implementation

- Array implementation : Node numbered k is stored in an array tree[k].
- Linked Implementation : Struct Node value, left, right

Tree Traversal

Tree traversal is passing through the tree and visiting each node exactly once (linearization). There are three ways to traverse the tree. (preorder, inorder, postorder). All of tree traversal is variant of depth-first search(DFS) complemented by or recursive function call. Exceptionally, level-order traversal is breadth-first search(BFS) complemented by queue.

- Preorder(T) = <n, Preorder(T_1), Preorder(T_2), ..., Preorder(T_k)>
- Inorder(T) = <Inorder(T_1), n, Inorder(T_2), ..., Inorder(T_k)>
- Postorder(T) = <Postorder(T_1), Postorder(T_2), ..., Postorder(T_k), n>

Two type of traversal :

- Depth-first traversal
  Go down first, recursive way, 3 variation : preorder, inorder, postorder
- Breadth-first traversal
  Go across first in same level, No variation, just level-order traversal

For getting Unique binary tree by two traversals, only 3 types are valid : **(postorder and inorder), (preorder and inorder), (levelorder and inorder)**. Inorder are necessary to find left and right child, and postorder/preorder/levelorder are necessary to find root.

Some type of general tree implementations. This is example. **Simple Approach**

List-of-Children Approach
Left-Child/Right-Sibling Approach
TODO

Converting into a Binary Tree : Donald Knuth

Input : general trees. Then leftmost child is left child, and right sibling is right child. Finally, remove the other links. It seems like rotating the tree clockwise by 45deg. When going from parent to leftchild, level be increase by 1 from original tree, and when going from parent to rightchild, level be same as original tree. Preorder is same, Postorder of original will be Inorder of new binary tree.

6 Priority Queue & Heap

Priority Queue

Definition

Priority queue consists of a set of elements (organized by priority, also called key).
For implement priority queue, there are obvious ways.

	Insert	DeleteMin
Normal queue	$O(1)$	$O(n)$
Unsorted linked list	$O(1)$	$O(n)$
Sorted linked list	$O(n)$	$O(1)$

$O(n)$ seems to much... so we need to find a better way -¿ Heap!

Heap

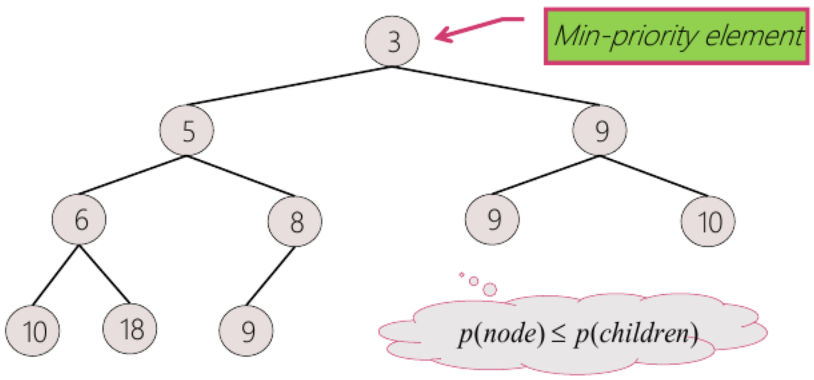
heap property

- if B is a child node of A, then $p(A) \leq p(B)$.
- Implies that an element with the lowest priority is always in the root node (mean-heap) $\leftrightarrow$ (max-heap)

  To efficiently implenet to priority queue -¿ Insert and DeleteMin : $O(\log n)$
  There are some types of heap : binary, binomial, fibonacci, 2-3, etc.

Binary Heap

Binary Heap satisfying two properties. (1) Complete binary tree (structural property) (can be implemented in an array), (2) Min tree (Heap order property) (p(node) $\leq$ p(children)) We call Complete Binary Tree as CBT.



- Insert
- DeleteMin

  Change Min Heap to Max Heap

7 Sorting

WOw! THIS already exists in SUMMARY! Note that.
In this part, It will show how elements can be sorted by each methods simply.

- Bubble Sort
- Selection Sort
- Insertion Sort
- Bucket Sort
- Merge Sort
- Quick Sort
- Heap Sort...?

8 Binary Search Tree

Definition

- Insert
- Delete
- Search

9 AVL Tree

Why use AVL tree?
  Balance factor
  Rotatie

- Insert
  - Unbalance - LL
  - Unbalance - RR
  - Unbalance - LR
  - Unbalance - RL
- Delete
- Search

TIP!

- Mergesort, quicksort are slower than Insertion sort when approximately $n \leq 15$
- When running quicksort, the ideal case is choosing the median key value.
  - Use the first or last element, then if the input is sorted, it will be poor.
  - Pick an element at random, but using a random number generator is expensive.
  - So, we have to select the middle element.
  - Using **median-of-three** rule : choose the median of the first, middle, and last element.
- Comparing Quicksort vs Heapsort : Quicksort is faster than heap, however quicksort should never be used in application which require a guarantee of response time unless it is treated as an $O(n \log n)$ algorithm.
- Comparing Quicksort vs Mergesort : Quicksort can be implemented in-place, but mergesort can't. No additional memory is required as in Merge sort. In many cases, quicksort can avoid $O(n^2)$ by choosing a right pivot.

Stable Sort : insertion, bubble,merge, couting, bucket
Unstable Sort : quick, heap, shell, selection

Comparison sort : bubble, selection, insertion, heap, merge, quick
non-comparison sort : counting, bucket, radix

10 Summary

Datastructure

Datastructure	Insert	Delete	Locate	Description
Array-based list	$O(n)$	$O(n)$	$O(n)$	If you know the proper pointer when insert, then $O(1)$
Pointer-based list	$O(1)$	$O(1)$	$O(n)$	
Cursor-based list	$O(1)$	$O(1)$	$O(1)$	
Array-Based Queue	$O(1)$	$O(1)$	$O(n)$	
Linked Queue	$O(1)$	$O(1)$	$O(n)$	
Tree	$O(n)$	$O(n)$	$O(n)$	According to implementation types, it can be changed. Check some approach of trees.
Priority Queue	$O(logn)$	$O(logn)$	$O(n)$	
Heaps	$O(logn)$	$O(logn)$	$O(n)$	
Binary Search Tree	$O(logn)$	$O(logn)$	$O(logn)$	worst case, $O(n)$
AVL tree	$O(logn)$	$O(logn)$	$O(logn)$	

Algorithm

Algorithm	Worst	Average	Best	Description
Bubble sort	$O(n^2)$	$\Theta(n^2)$	$\Omega(n)$	Comparison, Stable Sort
Insertion sort	$O(n^2)$	$\Theta(n^2)$	$\Omega(n)$	Comparison, Stable Sort
Selection sort	$O(n^2)$	$\Theta(n^2)$	$\Omega(n^2)$	Comparison, Unstable Sort
Bucket sort	$O(n^2)$	$\Theta(n+N)$	$\Omega(n+N)$	Non-Comparison, Stable Sort. N is the number of buckets. Worst is that all is in only one.
Merge sort	$O(n\log n)$	$\Theta(n\log n)$	$\Omega(n\log n)$	Comparison, Stable Sort, Divide-and-Conquer Recursive(up&down) -or, Non-recursive(upward)
Quicksort	$O(n^2)$	$\Theta(n\log n)$	$\Omega(n\log n)$	Comparison, Unstable Sort, Worst is that pivot selection -is not balanced.
Heap sort	$O(n\log n)$	$\Theta(n\log n)$	$\Omega(n\log n)$	Comparison, Unstable Sort