Hash Tables [5/5]

Efficiency Comparison (duplicate keys not allowed)

	Idea #1	Idea #2	Idea #3	
	Priority Queue using Heap	Self-Balancing Search Tree	Hash Table: worst case	Hash Table: average case
Retrieve	Constant*	Logarithmic	Linear	Constant
Insert	Amortized** logarithmic	Logarithmic	Linear	Amortized constant***
Delete	Logarithmic*	Logarithmic	Linear	Constant

*Priority Queue *retrieve* & *delete* are not Table operations in full generality. Only the item with the highest priority (key) can be retrieved/deleted.

**Logarithmic if enough memory is preallocated. Otherwise, occasional reallocate-and-copy—linear time—may be required. Time per *insert*, averaged over many consecutive *inserts*, will be logarithmic. Thus, *amortized logarithmic time* (which is not a term I expect you to know).

***Hash Table *insert* is constant-time only in a *double average* sense: averaged both over all possible inputs and over a large number of consecutive *inserts*.

Position-Oriented ADT	**Corresponding Value-Oriented ADT**
Sequence	SortedSequence
Binary Tree	Binary Search Tree

	Sorted Array	Unsorted Array	Sorted Linked List	Unsorted Linked List	Binary Search Tree	*Strongly Balanced (how?) BST*
Retrieve	Logarithmic	Linear	Linear	Linear	Linear	*Logarithmic*
Insert	Linear	Constant-ish	Linear	Constant	Linear	*Logarithmic*
Delete	Linear	Linear	Linear	Linear	Linear	*Logarithmic*

Review
More on Linked Lists [2/5]

	Smart Array	Linked List
Look-up by index	$\boldsymbol{O(1)}$	$\boldsymbol{O(n)}$
Search sorted	$\boldsymbol{O(\log n)}$	$\boldsymbol{O(n)}$
Search unsorted	$O(n)$	$O(n)$
Sort	$O(n \log n)$	$O(n \log n)$
Insert @ given pos	$\boldsymbol{O(n)}$	$\boldsymbol{O(1)}$*
Remove @ given pos	$\boldsymbol{O(n)}$	$\boldsymbol{O(1)}$*
Splice	$\boldsymbol{O(n)}$	$\boldsymbol{O(1)}$
Insert @ beginning	$\boldsymbol{O(n)}$	$\boldsymbol{O(1)}$
Remove @ beginning	$\boldsymbol{O(n)}$	$\boldsymbol{O(1)}$
Insert @ end	$O(n)$** amortized const	$O(1)$ or $O(n)$***
Remove @ end	$O(1)$	$O(1)$ or $O(n)$***
Traverse	$O(n)$	$O(n)$

*For Singly Linked Lists, insert/remove just *after* the given position.

- Doubly Linked Lists can help.

**$O(1)$ if no reallocate-and-copy.

- Pre-allocate to ensure this.

***For $O(1)$, need a pointer to end of list. Otherwise, $O(n)$.

- This can be tricky.
- And, for remove @ end, it is mostly impossible.
- Doubly Linked Lists can help.

Find faster
with an array

Rearrange faster
with a Linked List

Review
More on Linked Lists [5/5]

	Smart Array	**Doubly** Linked List
Look-up by index	$\boldsymbol{O(1)}$	$\boldsymbol{O(n)}$
Search sorted	$\boldsymbol{O(\log n)}$	$\boldsymbol{O(n)}$
Search unsorted	$O(n)$	$O(n)$
Sort	$O(n \log n)$	$O(n \log n)$
Insert @ given pos	$\boldsymbol{O(n)}$	$\boldsymbol{O(1)}$
Remove @ given pos	$\boldsymbol{O(n)}$	$\boldsymbol{O(1)}$
Splice	$\boldsymbol{O(n)}$	$\boldsymbol{O(1)}$
Insert @ beginning	$\boldsymbol{O(n)}$	$\boldsymbol{O(1)}$
Remove @ beginning	$\boldsymbol{O(n)}$	$\boldsymbol{O(1)}$
Insert @ end	$O(n)$** amortized const	$O(1)$
Remove @ end	$O(1)$	$O(1)$
Traverse	$O(n)$	$O(n)$

Find faster
with an array

Rearrange faster
with a Linked List

With Doubly Linked Lists, we can eliminate asterisks.

**$O(1)$ if no reallocate-and-copy.

- Pre-allocate to ensure this.

Sequences in the C++ STL
Efficiency [2/2]

	`vector`, `basic_string`	`deque`	`list`
Look-up by index	Constant	Constant	Linear
Search sorted	Logarithmic	Logarithmic	Linear
Insert @ given pos	Linear	Linear	Constant
Remove @ given pos	Linear	Linear	Constant
Insert @ end	Linear/ Amortized constant*	Linear/ Amortized constant**	Constant
Remove @ end	Constant	Constant	Constant
Insert @ beginning	Linear	Linear/ Amortized constant**	Constant
Remove @ beginning	Linear	Constant	Constant

The way `vector` acts at the end is the way `deque` acts at beginning *and* end.

*$\Theta(1)$ if sufficient memory has already been allocated. We can pre-allocate.

**Only a constant number of value-type operations are required. The C++ Standard says these are constant-time.

All four have $\Theta(n)$ traverse & search-unsorted and $\Theta(n \log n)$ sort.

Queues
In the C++ STL — Operations

The `std::queue` interface for the various ADT operations:

ADT Operation	Implementation
enqueue	Member function `push`
dequeue	Member function `pop`
getFront	Member function `front`
isEmpty	Member function `empty`
size	Member function `size`
create	Default constructor
destroy	Destructor
copy	Copy/move operations

This one is different from `std::stack`, which has `top`.

`std::queue` also has:

- Member function `swap`.
- The various comparison operators (`==`, `<`, etc.).

Amortized Constant Time [4/4]

Using Big-*O*	**In Words**
$O(1)$	Constant time
$O(\log n)$	Logarithmic time
$O(n)$	Linear time
$O(n \log n)$	Log-linear time
$O(n^2)$	Quadratic time
$O(c^n)$, for some $c > 1$	Exponential time

Q. Where does *amortized constant-time* fit into the above list?

A. It does *not* fit into the list!

The above are all about the worst-case time required for a *single operation*; amortized constant-time is not.

Insert-at-end for a well written resizable array is amortized constant-time. It is also still linear time.

Review
Analysis of Algorithms

Using Big-*O*	In Words
$O(1)$	Constant time
$O(\log n)$	Logarithmic time
$O(n)$	Linear time
$O(n \log n)$	Log-linear time
$O(n^2)$	Quadratic time
$O(c^n)$, for some $c > 1$	Exponential time

Cannot read all of input ↑

Probably not scalable ↓

Faster ↑

Slower ↓

Useful Rules

- **Rule of Thumb.** For nested "real" loops, order is $O(n^t)$, where t is the number of nested loops.
- **Addition Rule.** $O(f(n)) + O(g(n))$ is either $O(f(n))$ or $O(g(n))$, *whichever is larger*. And similarly for Θ. This works when adding up any *fixed*, *finite* number of terms.

Review
Parameter Passing I/II [1/2]

Four methods for passing a parameter or returning a value are used in C++:

	By Value `U f(T x)`	**By Reference** `U & f(T & x)`	**By Reference-to Const** `const U & f(const T & x)`	**By Rvalue Reference** `U && f(T && x)`
Makes a copy	YES ☹	NO ☺	NO ☺	NO ☺
Allows for polymorphism	NO ☹	YES ☺	YES ☺	YES ☺
Allows implicit type conversions	YES ☺	NO ☹	YES ☺	YES ☺
Allows passing of:	Any copyable value ☺	Non-const Lvalues ☹?	Any value* ☺	Non-const Rvalues*

*Rvalues *prefer* to be passed by Rvalue reference.