Implementations of Abstract Data Types

An ADT (abstract data type) typically has several different implementations. If we need make use of an ADT in a program, we must decide which of these implementations is best for our purposed. There are three criteria we should use to make such a choice:

1. time complexity,
2. space complexity,
3. complexity of the code itself.

Stack

The two standard implementations of *stack* are the array and the linked list implementations.

- Both implementations use $O(1)$ time for each operation.
- The array implementation is slightly simpler, but you must decide the capacity of the stack in advance.
- The linked list implementation allows flexibility of size.

Here are some suggestions.

- If you know in advance a maximum size needed, and your program uses only one stack, use the array implementation.
- If you do not know a maximum size needed, use the linked list implementation.
- If you have several stacks in the program and you know a maximum size needed for each, but expect that most of them are small and you don't know which ones will be small in advance, use the linked list implementation to save space.
- If we use a linked list implementation, and if items are continually inserted and deleted, we should make sure that we deallocate the dynamic memory when we delete items.

Queue

There are several implementations of the ADT *queue*. We have at least two different linked list implementations and at least three different array implementations. We want to make sure that we can execute each operation in $O(1)$ time.

Let M be the maximum number of items that will be in the queue at any one time, and let N is the total number of items that will ever be inserted. Obviously, $M \leq N$.

- If M is unknown, we should use a linked list implementation.
- If N is known and is not much larger than M, we should use the simple array implementation.

- If M is known and N is unknown or is much larger than M, we should use and array implementation. To handle *false overflow*, either use *slide*, meaning use an array of length $2N$ and copy items forward when the queue reaches the end of the array, or use a circular array of size at least $N+1$, where the first entry of the array is the successor of the last entry.
- If there are several queues in the program of varying sizes, and any one of them could have size up to N but most will be smaller, use a linked list implementation.
- If we use a linked list implementation, and if items are continually inserted and deleted, we should make sure that we deallocate the dynamic memory when we delete items.

Heap

Recall that a *heap* is a structure where any item may be inserted, but only the maximum (or the minimum) may be deleted.

- If the number of items in the heap is known not to exceed N, use the array implementation. The time needed for each operation (insert or delete max) is $O(\log n)$, where n is the current number of items.
- Dynamic binary tree implementation, not to be confused with a binary search tree. Each operation takes $O(\log n)$ time

Priority Queue

Stack, *queue*, and *heap* are all special cases of *priority queue*. That is a structure into which any item can be inserted, but only the item of highest priority can be deleted. For a stack, that item is the one most recently inserted, for a queue it is the one that has been there the longest, and for a max-heap it is the largest item. But priority could be defined in some other way.

Search Structure

A *search structure* permits any item to be inserted, and also permits any item to be found and possibly deleted. There are a number of implementations of this ADT.

- Array implementation of *unordered list*. The list is implemented as a prefix of an array of size N, where it is known that the structure will never have more than N items. An item can be inserted in $O(1)$ time. *Find* takes $O(n)$ time where n is the number of items in the list, since the only way to find an item is by *linear search*, but once it is found, it can be deleted in $O(1)$ time. This is the best implementation of *search structure* if there is just one search structure in the program, and the number of items is known to be very small. (If there are many search structures in the program, each expected to be very small, use an unordered linked list for each of them.)
- Linked list implementation of *unordered list*. *Find* still takes $O(n)$ time, where n is the number of items in the list, but insertion, and deletion of an item once found, both take $O(1)$ time. This implementation can be used if the maximum number of items is unknown, but is typically used if the number of items is

expected to be very small, such as for a bucket in a hash table that uses *closed addressing*, also known as *separate chaining*.

- Array implementation of *ordered list*. Insertion takes $O(n)$ time, since items larger than the item inserted must be slid to the right. Find takes $O(\log n)$ time using binary search, but deleting that item takes $O(n)$ time since items to the right must be slid to the left. Alternatively, we can use *lazy delete*, which takes $O(1)$ to delete an item which we have found.

- Dynamic implementation of *binary search tree*. That means each item is stored in a node which has pointers to its left and right children. Let h is the height of the tree and n the current number of items. Then $h = \Omega(\log n)$ but $h = O(n)$. Insertion, find, and delete each take $O(h)$ time. In most practical situations $h = O(\log n)$.

- Balanced binary search tree. To ensure that $h = O(\log n)$, we can use a *self-balancing* binary search tree. Three of these are *AVL tree*, *red-black tree*, and *treap*. However, a treap uses randomization and is only balanced with very high probability.

- B-tree. There are many forms of *B-trees*, including *2-3 trees*. For a *dictionary*, a B-tree is probably the best choice.

- Hash table. Each item has a *hash value* computed by a *hash function* H, and an item x is stored in an array at the address $H(x)$. There could be *collisions*, since $H(x) = H(y)$ is possible even if $x \neq y$. In that case, we need *collision resolution*. In open addressing, the array location stores just one item, but in case of a collision, an item could be stored in a different location. In closed addressing, the array location stores the set of all items with that hash value. We will learn about *open addressing*, *closed addressing*, *perfect hashing*, and *cuckoo hashing*. Confusingly, *closed hashing* uses open addressing, also called *probing*, rather than closed addressing.

  An example of a hash function used in practice is to take the last four digits of your social security number. Phyllis used the last four digits of a customer's telephone number as a hash value.

Question

A graph can be defined by the set of out-neighbors of each vertex. Which of the following do you think is best for representing a graph?

1. An array of arrays.
2. An array of linked lists.
3. A linked list of arrays.
4. A linked list of linked lists.