CIS 5511. Programming Techniques

Linear Structures and Hash Tables

1. Dynamic sets

Many data structures can be seen abstractly as *dynamic sets*, containing elements which can be moved into and out of them.

A dynamic set *S* usually supports the following "dictionary operations":

- *Insert(S, x)*: An operation that adds a new element pointed by *x* into *S*.
- *Delete(S, x)*: An operation that removes an element pointed by *x* from *S*.
- *Search(S, k)*: A query that locates an element in *S* with a given key *k*.

When there is an order defined among the elements, the following query operations are also often supported:

- *Minimum(S)*: Locate the element in *S* with the smallest key.
- *Maximum(S)*: Locate the element in *S* with the largest key.
- *Successor(S, x)*: Locate the element in *S* that is after *x* in the order.
- *Predecessor(S, x)*: Locate the element in *S* that is before *x* in the order.

The query operations do not change the content of the set, and may return a *NIL* pointer if there is no element in *S* satisfying the search condition. The *Insert* and *Delete* operations change the content of the data structure.

When implemented, a data structure represents the relation/order among objects in a domain as relation/order among objects within a computer. Different data structures handle the above operations differently. There are two common ways to represent a relation in computer:

- Implicitly, using the sequential order among memory cells,
- Explicitly, using specific pointers/references.

2. Array and linked-list

If the set is totally ordered (usually by key, but not always), then the data structure is *linear*, in the sense that every element in it has exactly one predecessor and one successor, except the minimum (which has no predecessor) and the maximum (which has no successor).

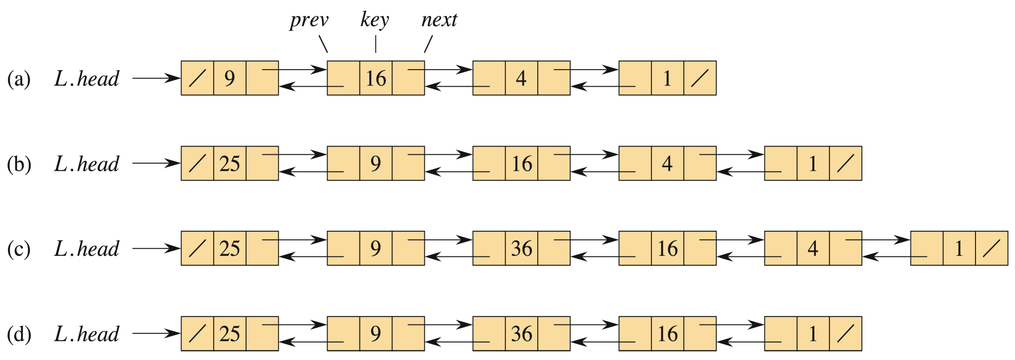
For a linear data structure, the order among elements is implicitly represented in an array, and explicitly in a linked list. Between the two, an array is more efficient in access at any position ("random access"), while a linked list is more efficient in insertions and deletions at a given position.

The elements of a linear structure can be data structures, such as in a matrix. In an Object-Oriented Programming language, linked lists are often implemented in two ways, with or without a specific "head" object containing the reference to the first node. Without such a head, the "node" class and the "list" class are merged, and the list is defined recursively as either empty or a node succeeded by a list.

A linked list can also be implemented in two arrays, one of which keeps the elements themselves, and the other keeps the index of the successor for each element. Similarly, multiple arrays can be used to represent an array of objects with multiple fields.

To turn an array of length n into a circular array, use *index = (index mod n) + 1*. In a linked list, let the last element point to the first element.

To move in both directions in a linked list, an additional link can be used to turn a *singly-linked list* into a *doubly-linked list*.



List operations:

```
LIST-SEARCH(L, k)
1  $x = L.head$
2  while $x \neq$ NIL and $x.key \neq k$
3      $x = x.next$
4  return $x$

LIST-INSERT(x, y)
1  $x.next = y.next$
2  $x.prev = y$
3  if $y.next \neq$ NIL
4      $y.next.prev = x$
5  $y.next = x$

LIST-DELETE(L, x)
1  if $x.prev \neq$ NIL
2      $x.prev.next = x.next$
3  else $L.head = x.next$
4  if $x.next \neq$ NIL
5      $x.next.prev = x.prev$
```

3. Stack and queue

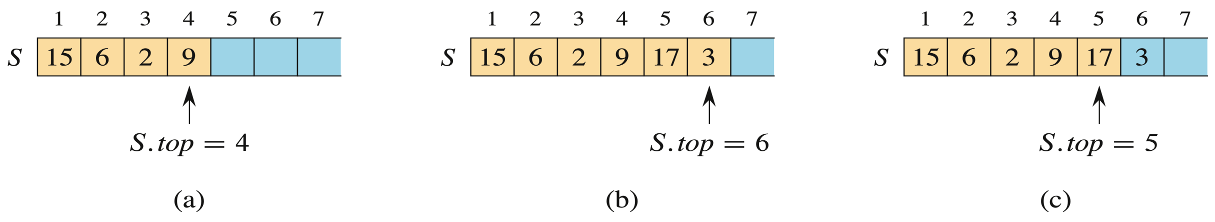
Stack and queue are abstract data structures where the order to be maintained is the insertion order of element to the structure. It is assumed that the deletion order is also the same (in queues) or the reverse (in stacks) of the insertion order, therefore both insertion and deletion become "zero address" operations — the position where the operation is performed is always at the first or last element, and any access to the other elements are not allowed.

Insert/Delete on stack and queue are taken as Θ(1) operations, and the size of the data structure automatically grows when needed.

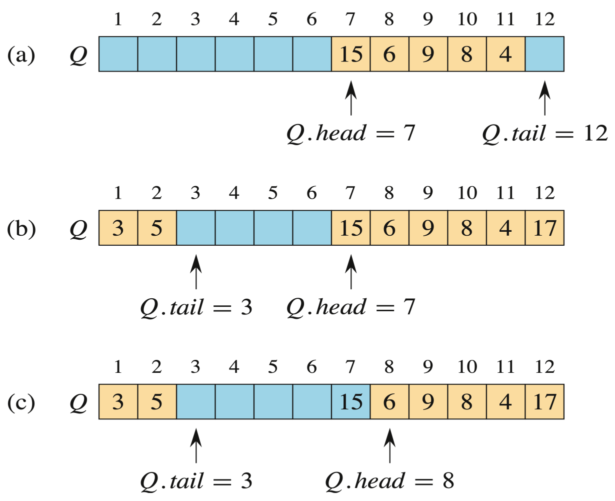
Both stack and queue can be implemented by array and linked list:

	STACK	QUEUE
array	operations happen at the highest index value	a circular array is needed to work on both ends
linked list	a reference points to the last node	two references points to both end, and in a singly-linked list, deletion can only happen at one end

Array-Stack:



Array-Queue:



4. Hash table

Conceptually, a hash table is a dynamic set without internal order. It provides quick search/insert/delete by directly mapping a key value to an index in an array (table), but does not efficiently support the operations defined with respect to inter-element order or relation.

Among the three operations, each insertion and deletion usually requires a search (as duplicate keys are not allowed), therefore search is the representative operation in efficiency analysis. Since search is normally a mapping from a *key* to an *index* in the table, the intuition behind hash table is to directly build such a mapping *index = h(key)*, without comparing the key with the elements in the table. Since the range of index is much smaller than the range of key, the mapping is many-to-one, not one-to-one.

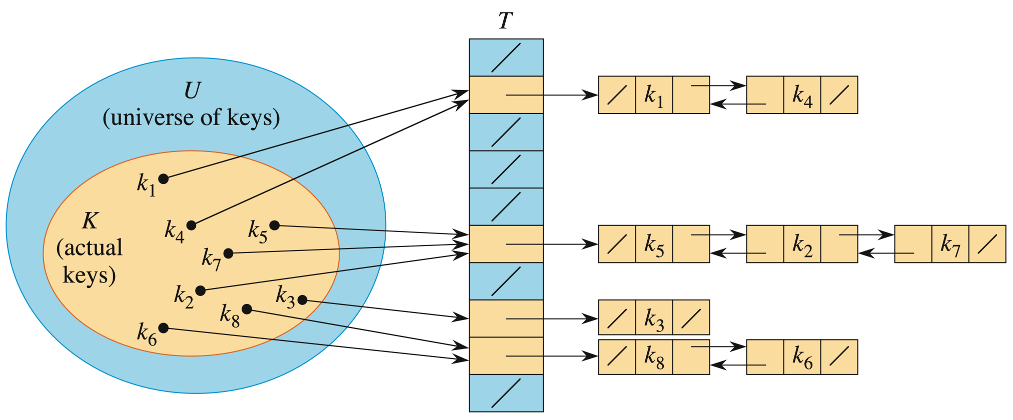
The design of a hash table consists of two major decisions: (1) to define a hash function *index = h(key)*, (2) to handle "collision", the situation where multiple keys are mapped into the same index.

Typically, the size of a hash table is proportional to the number of keys actually stored, while being much smaller than the range of possible keys.

A good hash function satisfies (approximately) the assumption that each key is equally likely to hash to any of the *m* index values, independently of where any other key has hashed to. Often, a hash function works in two steps: (1) to convert a key into an integer, (2) to calculate the index from the integer. Therefore, hash function discussion often assumes that the function is a mapping from an integer (key) to an integer (index).

The most common hash function is to take the reminder of the key divided by the size of the hash table, that is, *h(k) = k mod m*, assuming index in [0, m – 1]. A more complicated version is *h(k) = f(k) mod m*, where *f(k)* does additional calculation to reduce collision, and the reminder operation makes the function value to cover the whole table. A similar approach is *h(k) = floor(g(k) * m)*, where *g(k)* maps *k* into the range of [0, 1).

Common collision handling methods can be divided into two types, *open addressing*, where all elements are stored in the table itself, and *separate chaining*, where elements are stored outside the table in (sorted) linked-lists ("buckets"). Separate chaining requires additional space, though is conceptually simpler than open addressing.



In open addressing, the hash function generates a probe sequence to tell the element where to go if the slot indicated by the hash function is already occupied by a different key. Such a sequence can depend on the key (such as double hashing) or follow a fixed pattern (such as linear probing and quadratic probing). In either way, element comparisons are necessary, as collision can happen in multiple places. After deletion, the released space is often marked for the following search to pass through. Since insertion includes search, a new key can be stored in a previously occupied space only after the searching stops at a never-occupied space somewhere down the path. Another strategy is lazy deletion that relocates some element in a following search. To make things simple, we can also only insert into an empty cell, and let the "previous-occupied" marks be cleared periodically by re-hashing all elements.

Under the assumption that each possible probing sequence is equally probable, and that the load factor of the hash table α (number of elements / table size) is less than 1, there are the following conclusions for open addressing:

- Theorem 11.6: The expected number of probes in an unsuccessful search is at most 1/(1 – α).
- Corollary 11.7: The expected number of probes in an insertion is at most 1/(1 – α).
- Theorem 11.8: The expected number of probes in a successful search is at most (1/α) ln (1/(1 – α)).

Therefore, the average cost of the major operations of a hash table is Θ(1), though the worst-case cost is still Θ(*n*).