Elementary Data Structures

"Mankinds's progress is measured by the number of things we can do without thinking."

Elementary data structures such as stacks, queues, lists, and heaps will be the "of-the-shelf" components we build our algorithm from. There are two aspects to any data structure:

- The abstract operations which it supports.
- The implementatiton of these operations.

The fact that we can describe the behavior of our data structures in terms of abstract operations explains why we can use them without thinking, while the fact that we have different implementation of the same abstract operations enables us to optimize performance.

Stacks and Queues

Sometimes, the order in which we retrieve data is independant of its content, being only a function of when it arrived.

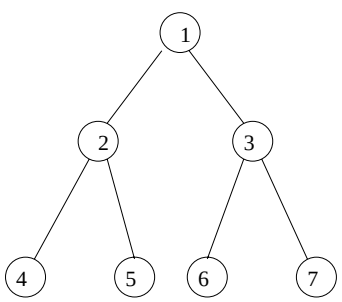
A *stack* supports last-in, first-out operations: push and pop.

A *queue* supports first-in, first-out operations: enqueue and dequeue.

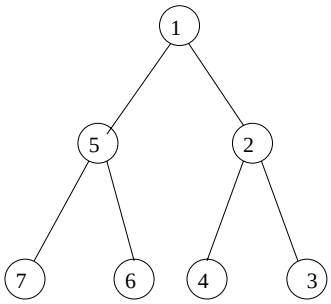
A *deque* is a double ended queue and supports all four operations: push, pop, enqueue, dequeue.

Lines in banks are based on queues, while food in my refrigerator is treated as a stack.

Both can be used to traverse a tree, but the order is completely different.



Queue



Stack

Which order is better for WWW crawler robots?

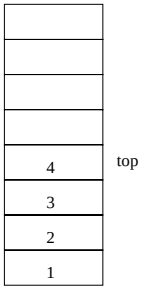
Stack Implementation

Although this implementation uses an array, a linked list would eliminate the need to declare the array size in advance.

STACK-EMPTY(S)
    if top[S] = 0
        then return TRUE
        else return FALSE

PUSH(S, x)
    $top[S] \leftarrow top[S] + 1$
    $S[top[S] \leftarrow x$

POP(S)
    if STACK-EMPTY(S)
        then error "underflow"
        else $top[S] \leftarrow top[S] - 1$
            return $S[top[S] + 1]$

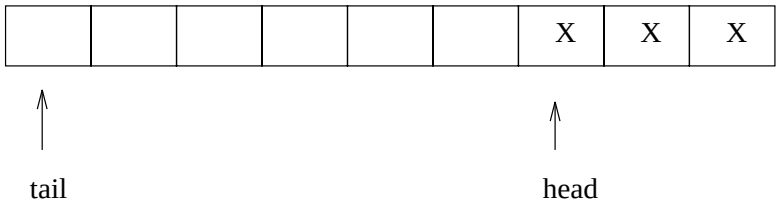


All are $O(1)$ time operations.

Queue Implementation

A circular queue implementation requires pointers to the head and tail elements, and wraps around to reuse array elements.

```
ENQUEUE(Q, x)
        Q[tail[Q]] ← x
        if tail[Q] = length[Q]
                then tail[Q] ← 1
                else tail[Q] ← tail[Q] + 1
```




```
DEQUEUE(Q)
        x = Q[head[Q]]
        if head[Q] = length[Q]
                then head[Q] = 1
                else head[Q] = head[Q] + 1
        return x
```

A list-based implementation would eliminate the possibility of overflow.

All are $O(1)$ time operations.

Dynamic Set Operations

Perhaps the most important class of data structures maintain a set of items, indexed by keys.

There are a variety of implementations of these *dictionary* operations, each of which yield different time bounds for various operations.

- *Search(S,k)* – A query that, given a set S and a key value k, returns a pointer x to an element in S such that $key[x] = k$, or nil if no such element belongs to S.

- *Insert(S,x)* – A modifying operation that augments the set S with the element x.

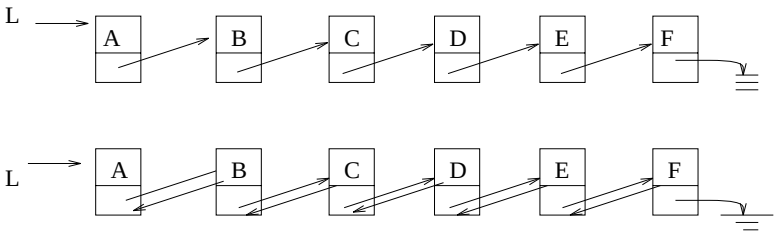
- *Delete(S,x)* – Given a pointer x to an element in the set S, remove x from S. Observe we are given a pointer to an element x, not a key value.

- *Min(S), Max(S)* – Returns the element of the totally ordered set S which has the smallest (largest) key.

- *Next(S,x), Previous(S,x)* – Given an element x whose key is from a totally ordered set S, returns the next largest (smallest) element in S, or NIL if x is the maximum (minimum) element.

Pointer Based Implementation

We can also maintain a directory in either a singly or doubly linked list.



We gain extra flexibility on predecessor queries at a cost of doubling the number of pointers by using doubly-linked lists.

Since the extra big-Oh costs of doubly-linkly lists is zero, we will usually assume they are, althought it might not be necessary.

Singly linked to doubly-linked list is as a Conga line is to a Can-Can line.

Array Based Sets

Unsorted Arrays

- Search(S,k) - sequential search, $O(n)$
- Insert(S,x) - place in first empty spot, $O(1)$
- Delete(S,x) - copy nth item to the xth spot, $O(1)$
- Min(S,x), Max(S,x) - sequential search, $O(n)$
- Successor(S,x), Predecessor(S,x) - sequential search, $O(n)$

Sorted Arrays

- Search(S,k) - binary search, $O(\lg n)$
- Insert(S,x) - search, then move to make space, $O(n)$
- Delete(S,x) - move to fill up the hole, $O(n)$
- Min(S,x), Max(S,x) - first or last element, $O(1)$
- Successor(S,x), Predecessor(S,x) - Add or subtract 1 from pointer, $O(1)$

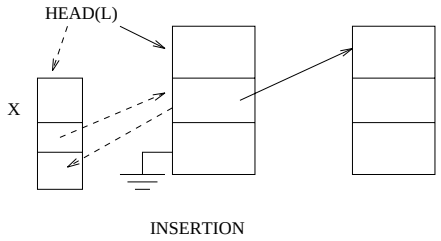
What are the costs for a heap?

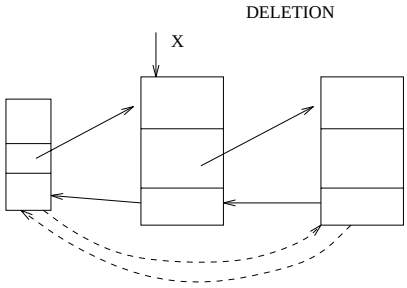
Unsorted List Implementation

LIST-SEARCH(L, k)

```
x = head[L]
while x <> NIL and key[x] <> k
      do x = next[x]
return x
```

Note: the while loop might require two lines in some programming languages.





LIST-INSERT(L, x)

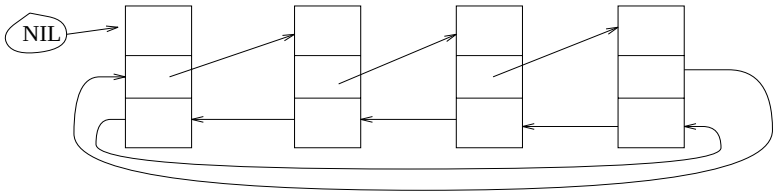
```
next[x] = head[L]
if head[L] <> NIL
      then prev[head[L]] = x
head[L] = x
prev[x] = NIL
```

LIST-DELETE(L, x)

```
if prev[x] <> NIL
      then next[prev[x]] = next[x]
      else head[L] = next[x]
if next[x] <> NIL
      then prev[next[x]] = prev[x]
```

Sentinels

Boundary conditions can be eliminated using a sentinel element which doesn't go away.



LIST-SEARCH'(L, k)

x = next[nil[L]]
while x <> $NIL[L]$ and $key[x]$ <> k
do x = next[x]
return x

LIST-INSERT'(L, x)

next[x] = next[nil[L]]
prev[next[nil[L]]] = x
next[nil[L]] = x
prev[x] = NIL[L]

LIST-DELETE'(L, x)

next[prev[x]] <> next[x]
next[prev[x]] = prev[x]