How to Represent Sets?

- Finite dynamic sets, to be more precise.

- Operations on these sets:
  - search,
  - insert,
  - delete,
  - find minimum,
  - find maximum,
  - successor,
  - predecessor…

Efficiently in function of the type of use of the set.
Particular Cases

- If only *insert*, *delete*, and *test membership*, then such a dynamic set is called a dictionary.

- Best way to implement a set depends on the needed operations.

- No perfect set for everything.
Examples of Dynamic Sets

- Heaps.
- Stacks, queues, linked lists.
- Hash tables.
- Binary search trees.
- Red-black trees (particular balanced binary search tree).
- In general, they use pointers.
Stacks and Queues

- Specify which element the **delete** operation removes:
  - stacks = LIFO (last-in, first-out)
  - queues = FIFO (first-in, first-out)
- **Insert** operation called **push** or **enqueue**.
- **Delete** operation called **pop** or **dequeue**.
- Can be implemented with an array.
- **Insert** and **delete** in $O(1)$. 
Stack Operations

```
stack_empty(S):  // test emptiness
    return top(S) == 0  // index of last element

push(S,x):
    top(S) = top(S)+1
    S[top(S)] = x

pop(S):
    if stack_empty(S) then error
    else
        top(S) = top(S)-1
        return S[top(S)+1]
    fi
```

Pseudo-code is abstract and does not address the issue of limited arrays [1..n].
Stack Operations

\[ \text{stack_empty}(S): \quad // \text{test emptiness} \]
\[ \text{return} \ \text{top}(S) == 0 \quad // \text{index of last element} \]

Empty: 0 element.

Not empty: 2 elements.
Stack Operations

push(S, x):
  top(S) = top(S) + 1
  S[top(S)] = x
Stack Operations

pop(S):
if stack_empty(S) then error
else
  top(S) = top(S) - 1
  return S[top(S) + 1]
fi

1 2 3 4 ...
S:          x
0 1 2 3 4 ...
top(S)

0 1 2 3 4 ...
S:          x
1 2 3 4 ...
top(S)

0 1 2 3 4 ...
S:          x
2 3 4 ...
top(S)

Error
Queue Operations

enqueue(Q, x):
  Q[tail(Q)] = x
  tail(Q) = tail(Q) + 1

dequeue(Q):
  x = Q[head(Q)]
  head(Q) = head(Q) + 1
  return x

Limited array:

enqueue(Q, x):
  Q[tail(Q)] = x
  if tail(Q) == length(Q)
    tail(Q) = 1
  else
    tail(Q) = tail(Q) + 1
  fi

dequeue(Q):
  x = Q[head(Q)]
  if head(Q) == length(Q)
    head(Q) = 1
  else
    head(Q) = head(Q) + 1
  fi
  return x
Queue Operations

Operations are circular, i.e., modulo the size:

enqueue(Q, x):
Q[tail(Q)] = x
tail(Q) = (tail(Q)+1) \% length(A) + 1

decqueue(Q):
x = Q[head(Q)]
head(Q) = (head(Q)+1) \% length(A) + 1
return x

Underflow/overflow not detected.
Empty/Full Queues

Choice:

Q: head(Q) tail(Q)

Full

Q: tail(Q) head(Q)

Empty

Q: head(Q) full=false/
tail(Q) size=0

Q: tail(Q) full=true/
head(Q) size=length(Q)
Empty/Full Queue

Q: [1, 2, 3, 4, 5]

head(Q) →

tail(Q)

Q: [1, 2, 3, 4, 5]

tail(Q) →

head(Q)

queue_empty(Q):
return head(Q) == tail(Q)

queue_next(Q, i):
return (i+1)%length(Q) + 1

queue_full(Q):
return
queue_next(Q, head(Q)) == tail(Q)
Queue Operations - Revisited

enqueue(Q,x):
  if queue_full(Q) then error
  Q[tail(Q)] = x
  tail(Q) = queue_next(Q,tail(Q))

dequeue(Q):
  if queue_empty(Q) then error
  x = Q[head(Q)]
  head(Q) = queue_next(Q,head)
  return x
Stacks/Queues

- In practice array [0..n-1], be careful.
- View stacks as bounded stacks and queues as pies.

Stack:
- top
- bottom

Queue:
- head
- tail
Linked Lists

- Linear structure, order given by pointers.
- Singly linked & doubly linked lists.
  - Singly linked = uni-directional.
  - Double linked = bi-directional.
- Lists = head + tail + elements of the list (typically called nodes = key + next + previous).

```
key next
<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
</table>
key|   |
|   |   |
prev| key| next
|   |   |
prev| key| next
|   |   |
prev| key| next
```
Lists - Search

**List_search(L,k):**

```plaintext
x = head(L)
while x != NIL and key(x) != k do
    x = next(x)
done
return x
```

NIL: special value, i.e., NULL pointer.

\(O(n)\) Returns NIL or the element \(x\) of the list s.t. key(x)==k.
Lists - Insert

List_insert1(L,x):
next(x) = head(L)
head(L) = x

List_insert2(L,x):
next(x) = head(L)
if head(L) != NIL then
    prev(head(L)) = x
fi
head(L) = x
prev(x) = NIL

Check: 2 updates.

$O(1)$
Lists - Insert

List\_insert2(L,x):

\[
\text{next}(x) = \text{head}(L) \\
\text{if } \text{head}(L) \neq \text{NIL} \text{ then} \\
\quad \text{prev}(\text{head}(L)) = x \\
\text{fi} \\
\quad \text{head}(L) = x \\
\quad \text{prev}(x) = \text{NIL}
\]

The tail is not used here. It can be equal to head, not a problem.

Check: 4 updates.

\(O(1)\)
Lists – Delete

Singly Linked List

Problem: You need to know where a node is referenced.

\[
\text{List}\_\text{delete}\_\text{first}(L) \\
\text{if} \ head(L) == \text{NIL} \ \text{then} \ \text{error} \\
\text{next} = \text{next}(\text{head}(L)) \\
\text{delete}(\text{head}(L)) \\
\text{head}(L) = \text{next}
\]

\[O(1)\]
List Delete

Doubly Linked List

List_delete(L, x):
if prev(x) != NIL then
    next(prev(x)) = next(x)
else
    head(L) = next(x)
fi

if next(x) != NIL then
    prev(next(x)) = prev(x)
fi
Linked Lists with Sentinels

- Sentinel = special element to avoid tests.
  - next(\texttt{nil}) = head(L), \ prev(\texttt{nil}) = tail(L)
  - empty list: next(\texttt{nil}) = prev(\texttt{nil}) = \texttt{nil}
- \texttt{nil} is the special element, it is \textit{not} \texttt{NIL}.
- Every list has its own \texttt{nil} sentinel.
- The list is now circular.

- Simplified algorithms.
  - Good for tight loops.
  - Bad if many small lists (memory overhead).
List Search with Sentinels

\[
\text{List\_search}(L,k): \\
x = \text{next}(\text{nil}) \\
\text{while } x \neq \text{nil} \text{ and } \text{key}(x) \neq k \text{ do} \\
\quad x = \text{next}(x) \\
\text{done} \\
\text{return } x
\]

head(L) NIL

Not much difference here.
List Delete with Sentinels

\[
\text{List}\_\text{delete}(L,x): \\
\text{next}(\text{prev}(x)) = \text{next}(x) \\
\text{prev}(\text{next}(x)) = \text{prev}(x)
\]

No if-statement.
List Insertion with Sentinels

**List insert**\((L, x)\):

1. `next(x) = next(nil)`
2. `prev(next(nil)) = x`
3. `next(nil) = x`
4. `prev(x) = nil`

No if-statement.

```
    prev | / | next
```

```
    prev | key | next
```
Coding with Arrays

- If you have no pointer, it is possible to use arrays and indices:
  - pointer (memory) $\leftrightarrow$ index (array).

- Used for specialized memory management:
  - one list of *used* elements,
  - one list of *free* elements.
Specialized Memory Management

Useful if

- many elements are allocated/de-allocated very often,
- you want to de-allocate everything and re-allocate again etc…

Allocate/de-allocate: update free and next(free). Commonly referred as “pool” – see C++ (Stroustrup).

Of course, initialize the list at the beginning!
Rooted Trees

- Trees represented by linked data structures.
  - Binary trees.
  - Trees with unbounded/dynamic branching.
  - Best representation depends on the application.
    - Heap: Intrinsic tree, no list.
Binary Trees
N-ary Trees
Doubly Linked Lists in C

typedef struct elem_s {
    struct elem_s *prev;
    struct elem_s *next;
    data_t key;
} elem_t;

typedef struct {
    elem_t *head;
} dlist_t;

or

typedef struct {
    elem_t nil;
} dlist_t;

void list_delete(dlist_t *l, elem_t *x)
{
    if (x->prev != NULL)
        x->prev->next = x->next;
    else
        l->head = x->next;
    if (x->next != NULL)
        x->next->prev = x->prev;
}

Special case for the head.
Variant of Doubly Linked Lists

typedef struct elem_s {
    struct elem_s **prev;
    struct elem_s *next;
    data_t key;
} elem_t;

typedef struct {
    elem_t *head;
} dlist_t;

void list_delete(elem_t *x) {
    *x->prev = x->next;
    if (x->next != NULL) {
        x->next->prev = x->prev;
    }
}

No special case for the head.