Data Structures

Alexandre David
B2-206

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
```

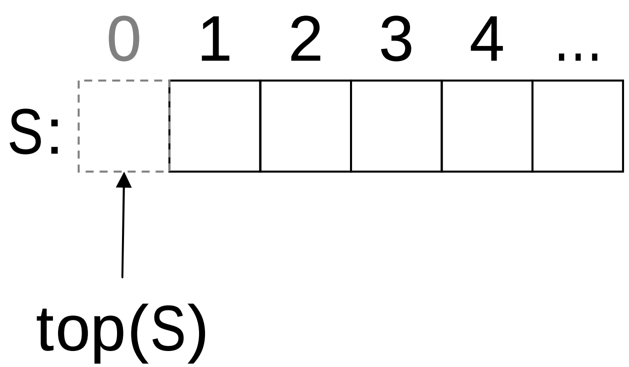
```
push(S,x):
top(S) = top(S)+1
S[top(S)] = x
```

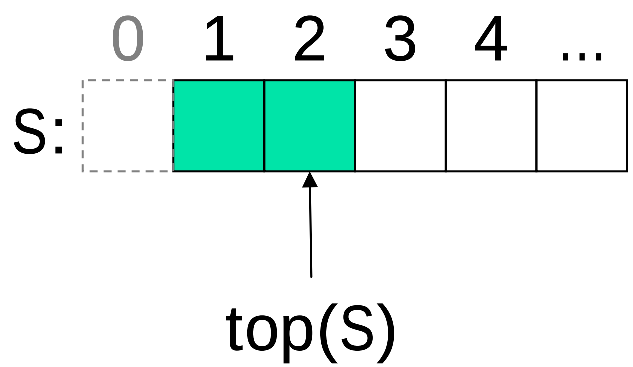
```
pop(S):
if stack_empty(S) then error
else
    top(S) = top(S)-1
    return S[top(S)+1]
fi
```

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

Stack Operations

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



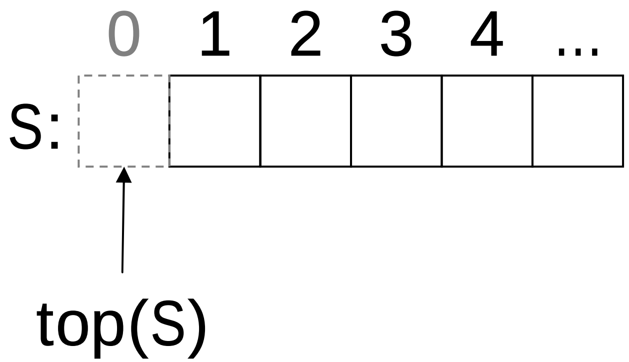


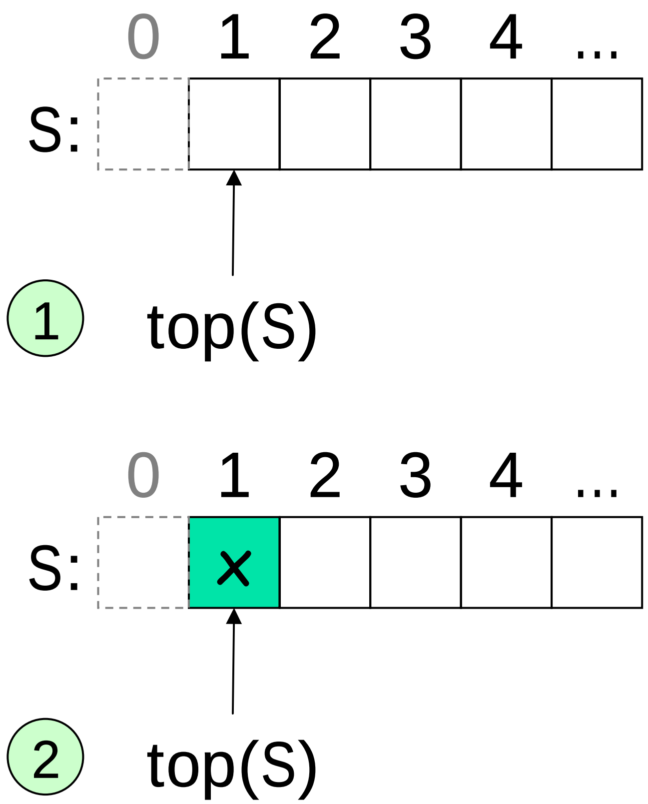
Empty: 0 element.

Not empty: 2 elements.

Stack Operations

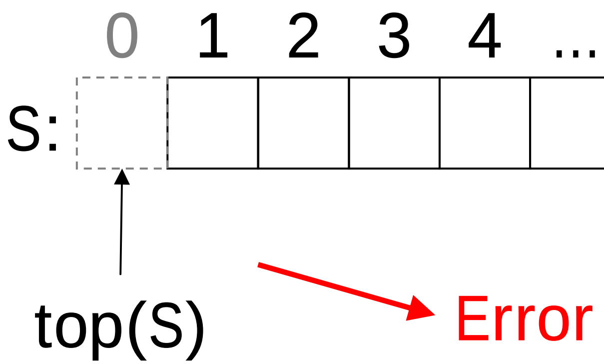
push(S,x):
1 top(S) = top(S)+1
2 S[top(S)] = x

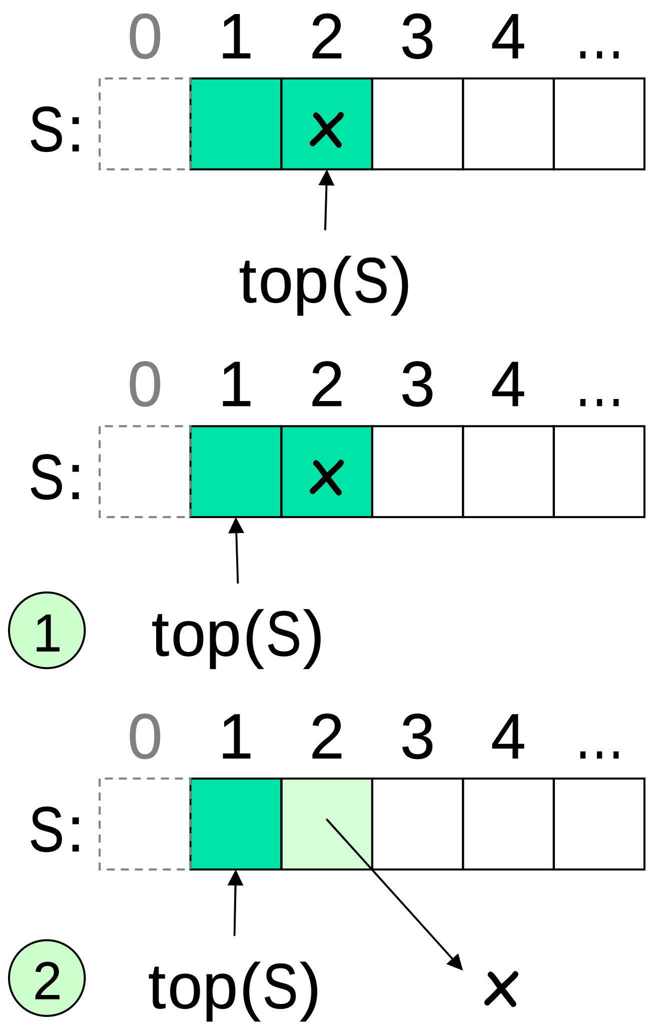




Stack Operations

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





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)
then
    tail(Q) = 1
else
    tail(Q) = tail(Q)+1
fi

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

Queue Operations

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

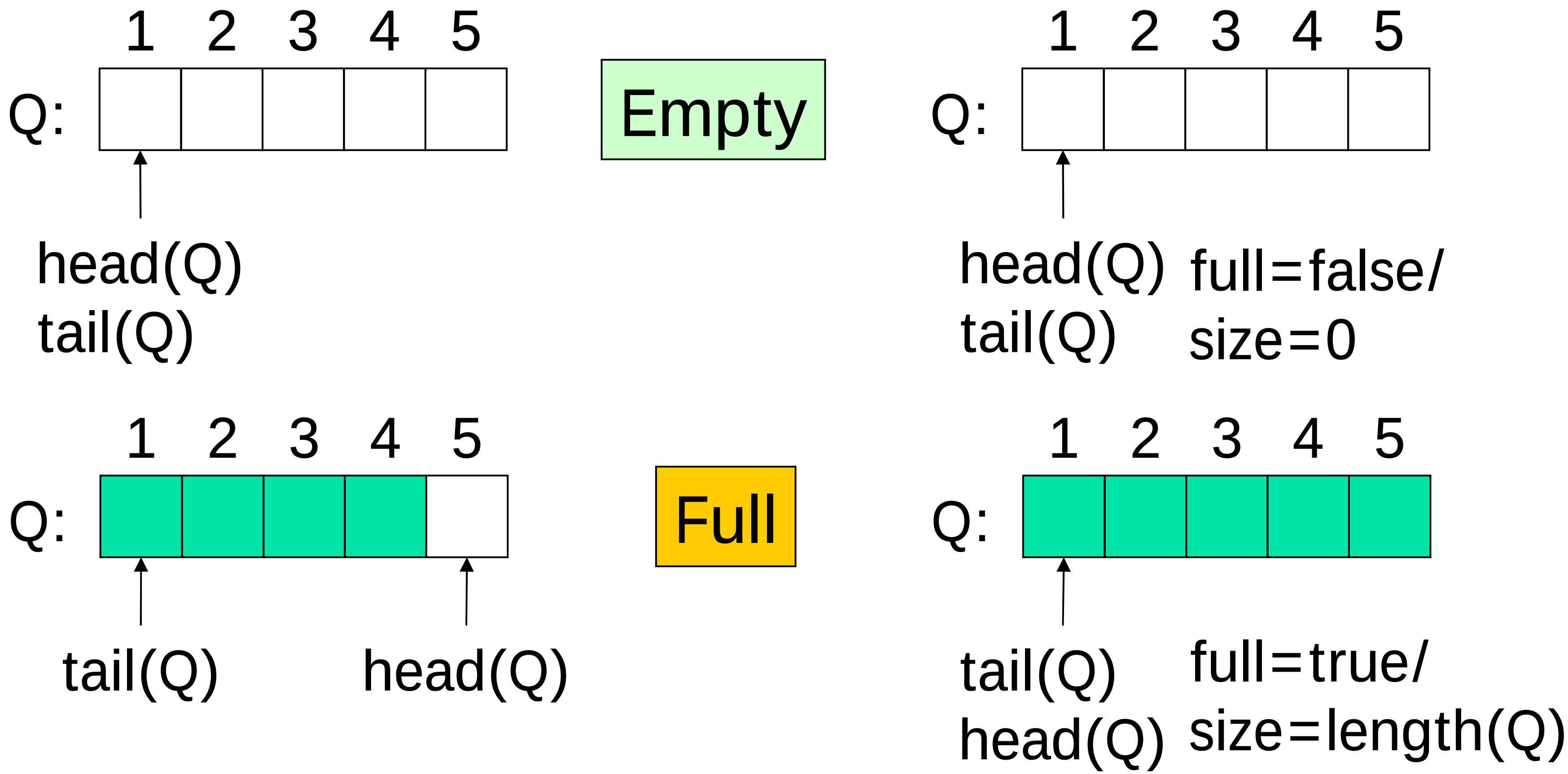
```
enqueue(Q,x):
Q[tail(Q)] = x
tail(Q) = (tail(Q)+1)%length(A) + 1
```

```
dequeue(Q):
x = Q[head(Q)]
head(Q) = (head(Q)+1)%length(A) + 1
return x
```

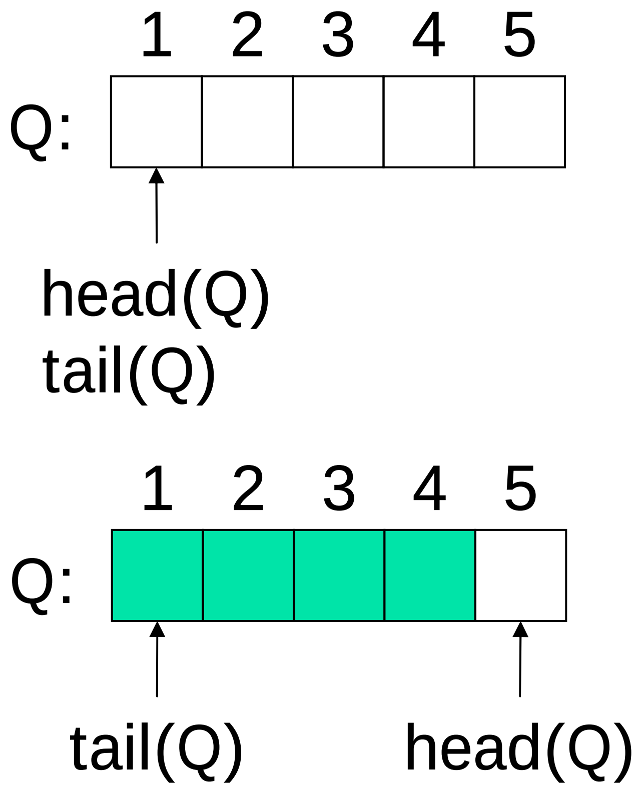
Underflow/overflow not detected.

Empty/Full Queues

Choice:



Empty/Full Queue




```
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)
```

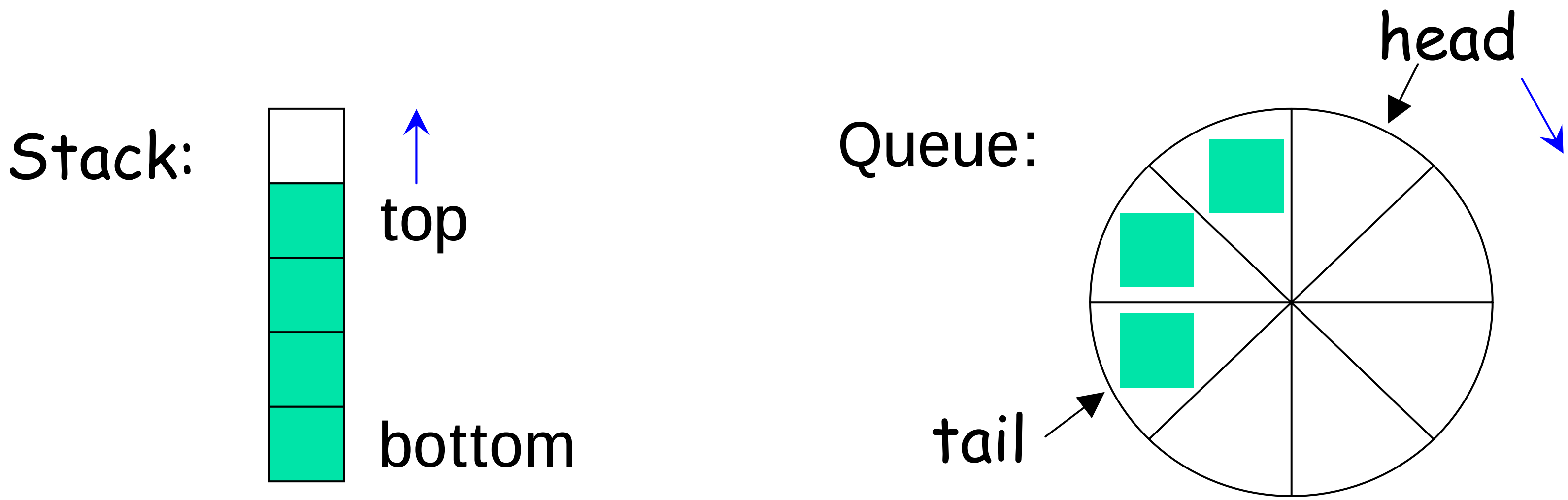
Queue Operations - Revisited

```
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
```

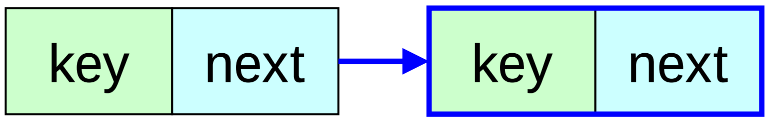
Stacks/Queues

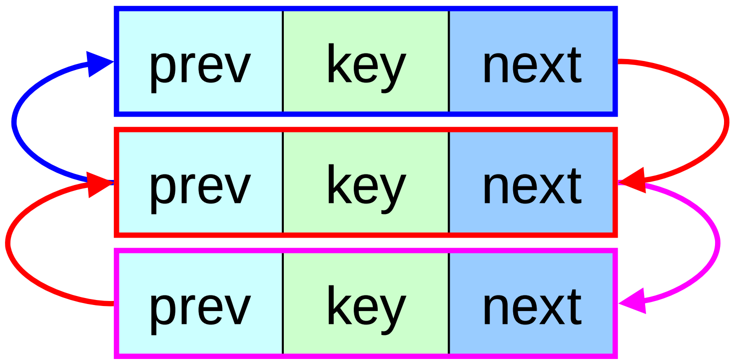
- In practice array [0..n-1], be careful.
- View stacks as bounded stacks and queues as pies.



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).





Lists - Search

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

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

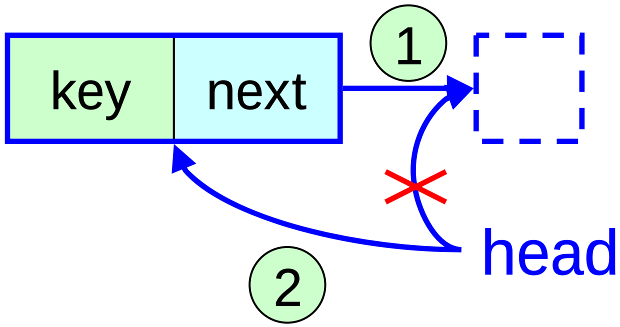
$\boxed{O(n)}$

Returns NIL or the element x of the list s.t. key(x)==k.

Make a drawing!

Lists - Insert

```
List_insert1(L,x):
1  next(x) = head(L)
2  head(L) = x
```



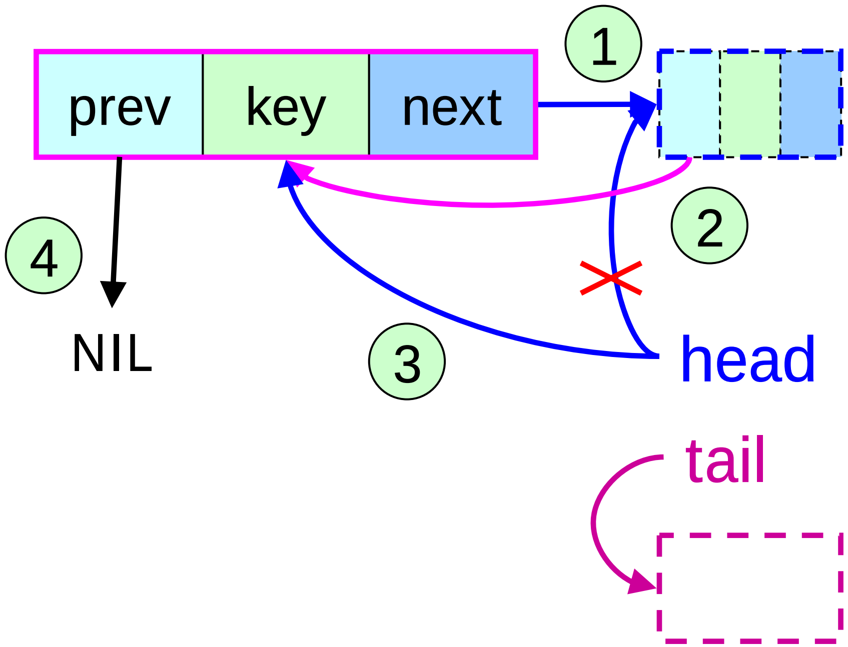
Check: 2 updates.

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

$O(1)$

Make a drawing!

Lists - Insert



List_insert2(L,x):
1 next(x) = head(L)
if head(L) != NIL **then**
2 prev(head(L)) = x
fi
3 head(L) = x
4 prev(x) = NIL

Check: 4 updates.

O(1)

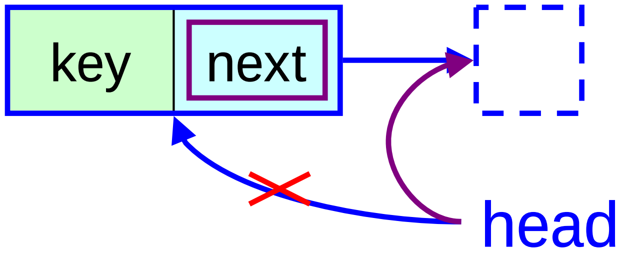
The tail is not used here.
It can be equal to head, not a problem.

Lists – Delete
Singly Linked List

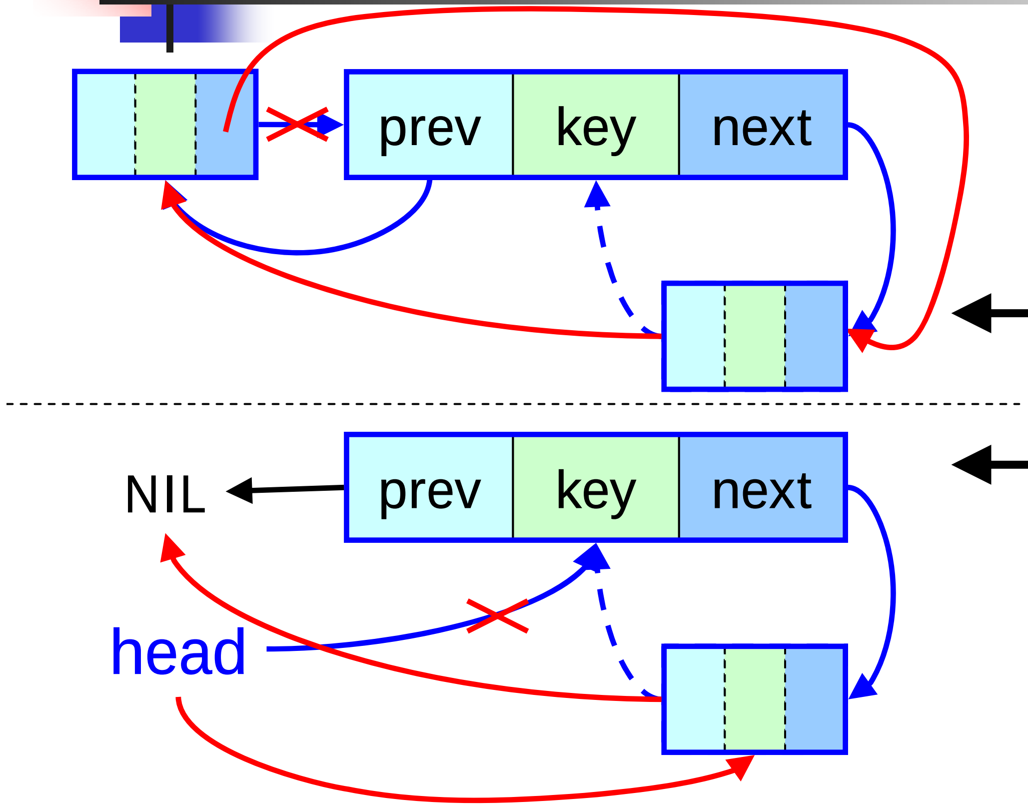
Problem: You need to know where a node is referenced.

```
List_delete_first(L)
if head(L) == NIL then error
next = next(head(L))
delete(head(L))
head(L) = 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
```

O(1)

Linked Lists with Sentinels

- Sentinel=special element to avoid tests.
  - next(**nil**)=head(L), prev(**nil**)=tail(L)
  - empty list: next(**nil**)=prev(**nil**)=**nil**
  - **nil** is the special element, it is <u>*not*</u> NIL.
  - Every list has its own **nil** sentinel.
  - The list is now circular.
- Simplified algorithms.
  - Good for tight loops.
  - Bad if many small lists (memory overhead).

List Search with Sentinels

```
List_search(L,k):
x = next(nil)
while x != nil and key(x) != k do
    x = next(x)
done
return x
```

head(L)
NIL

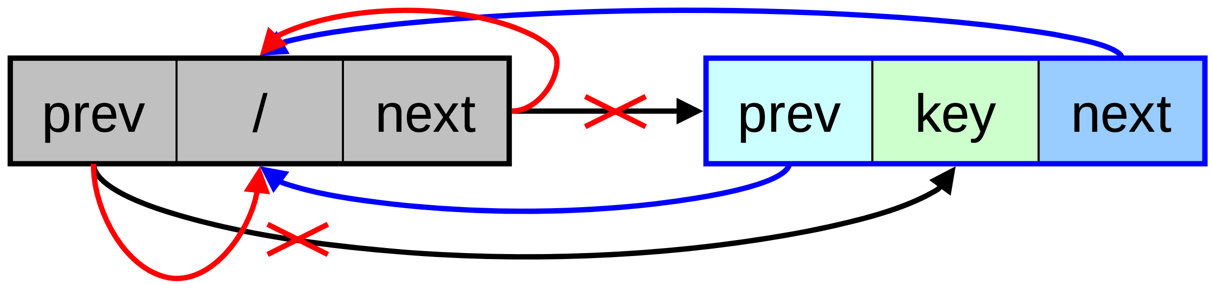
Not much difference here.

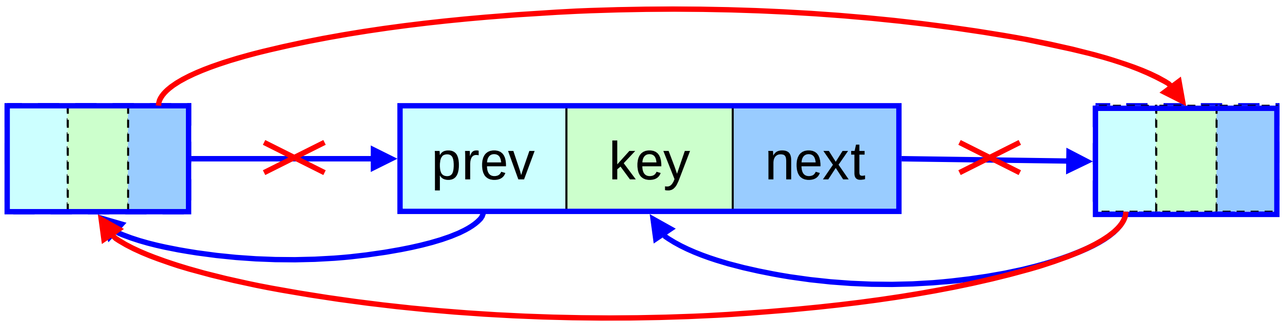
List Delete with Sentinels

```
List_delete(L,x):
next(prev(x)) = next(x)
prev(next(x)) = prev(x)
```

No if-statement.

nil(L):

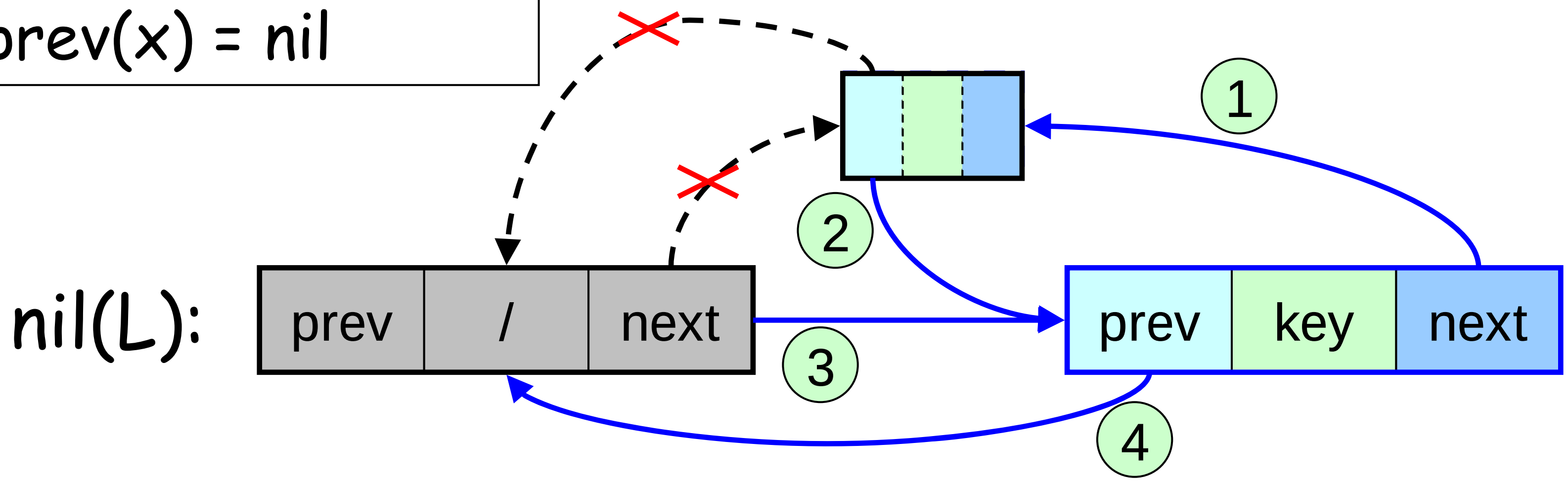




List Insertion with Sentinels

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.

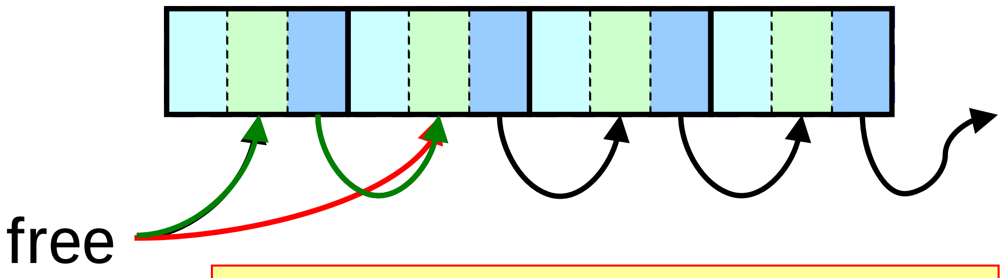


Coding with Arrays

- 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...



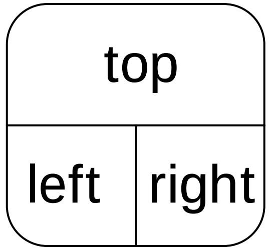
Of course, initialize the list at the beginning!

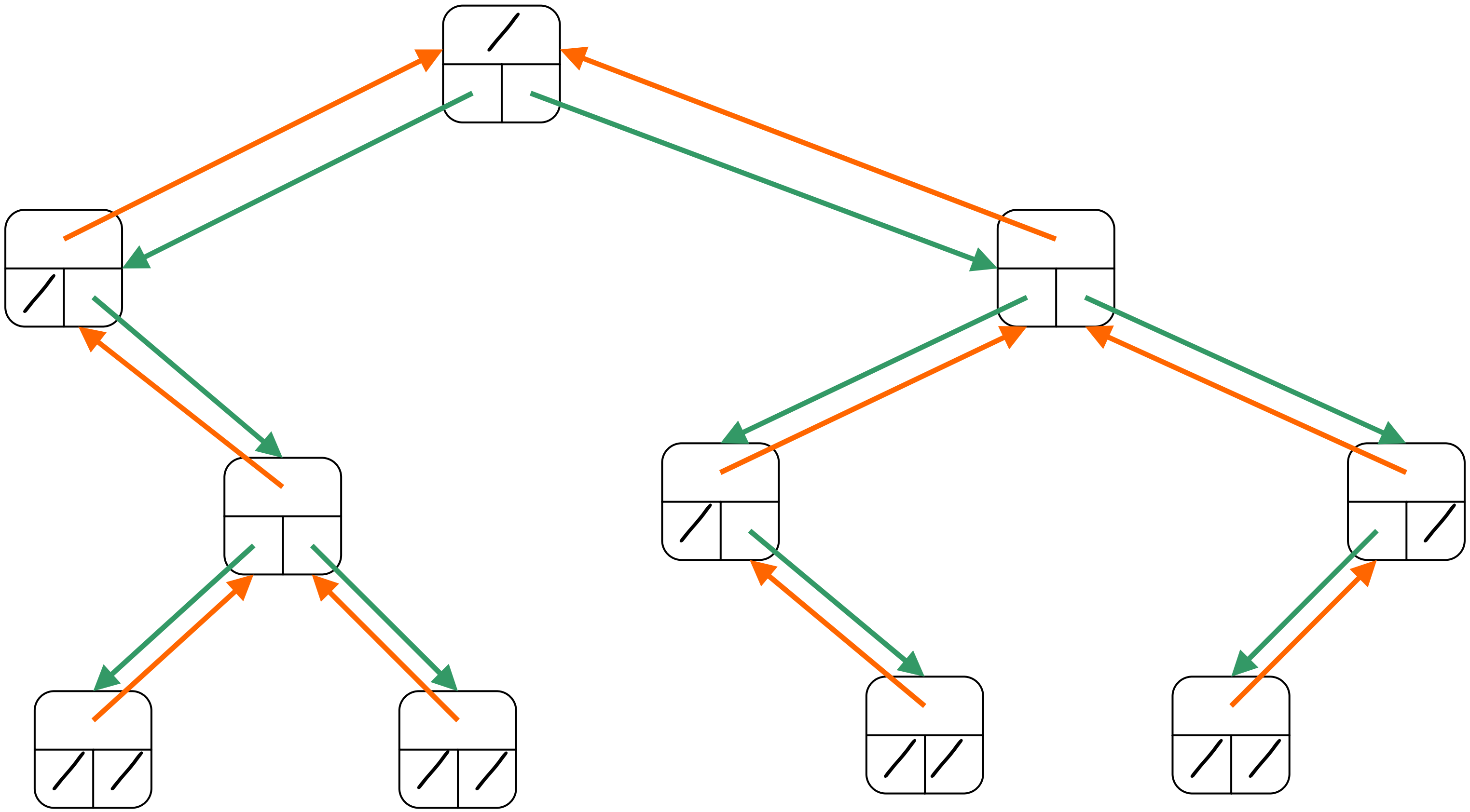
Allocate/de-allocate: update free and next(free). Commonly referred as “pool” – see C++ (Stroustrup).

Rooted Trees

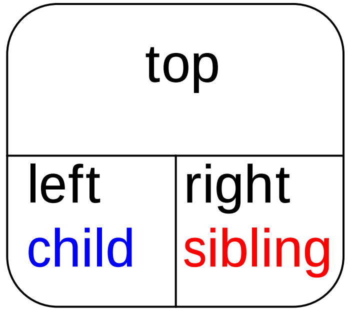
- Trees represented by linked data structures.
  - Binary trees.
  - Trees with unbounded/dynamic branching.
  - Best representation depends on the application.
    - Heap: Intrinsic tree, no list.

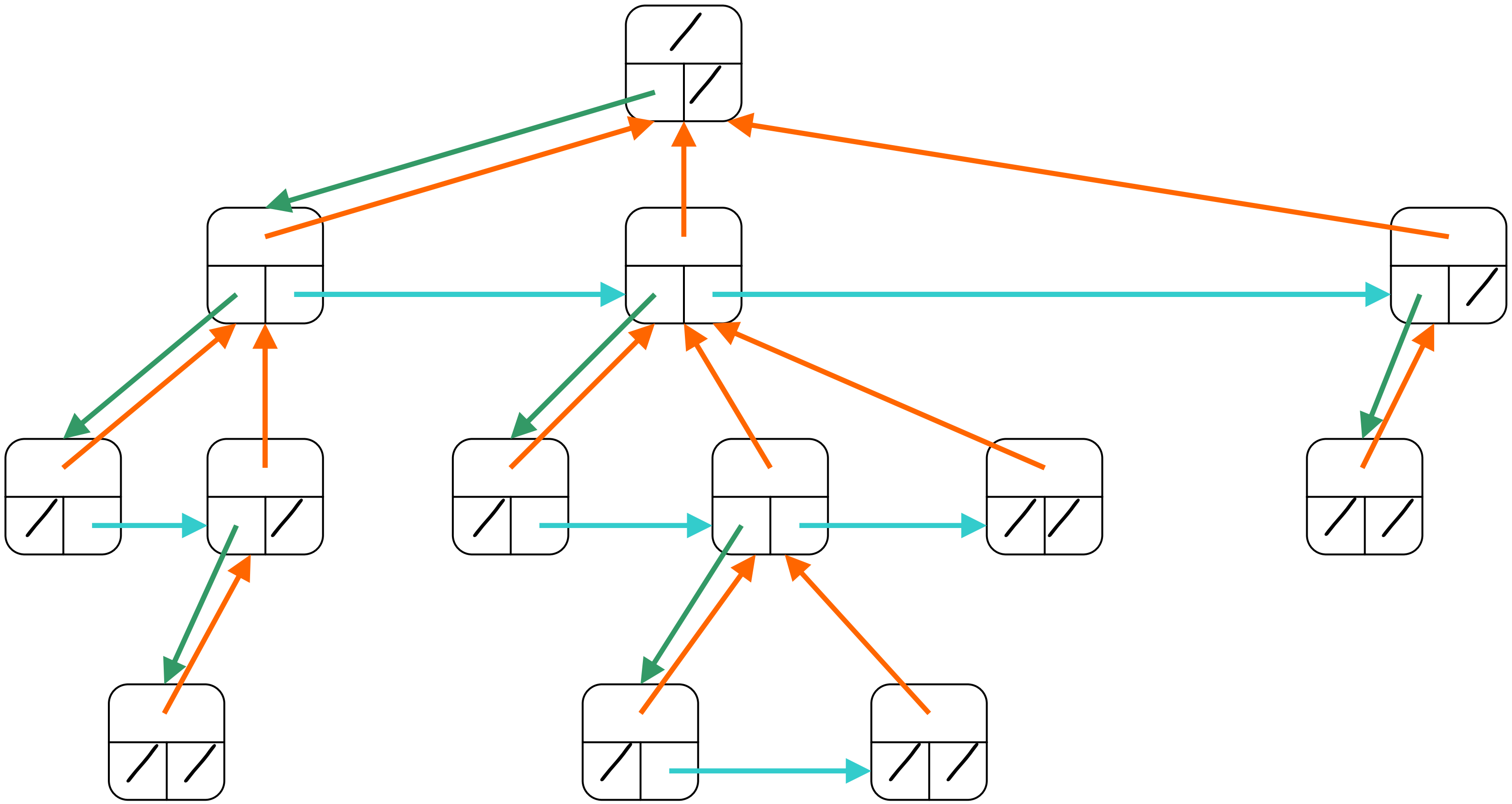
Binary Trees





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;
}
```

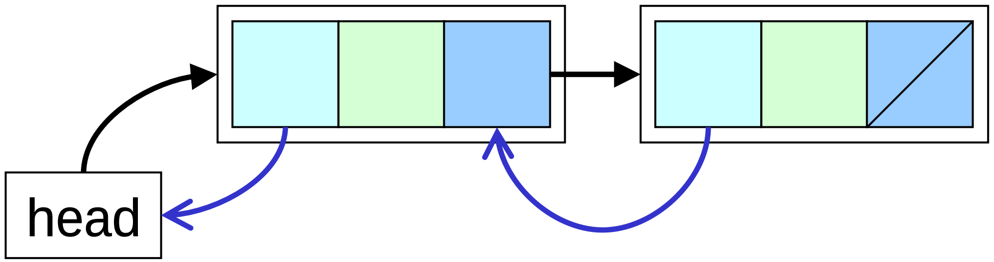
Special case for the head.

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.