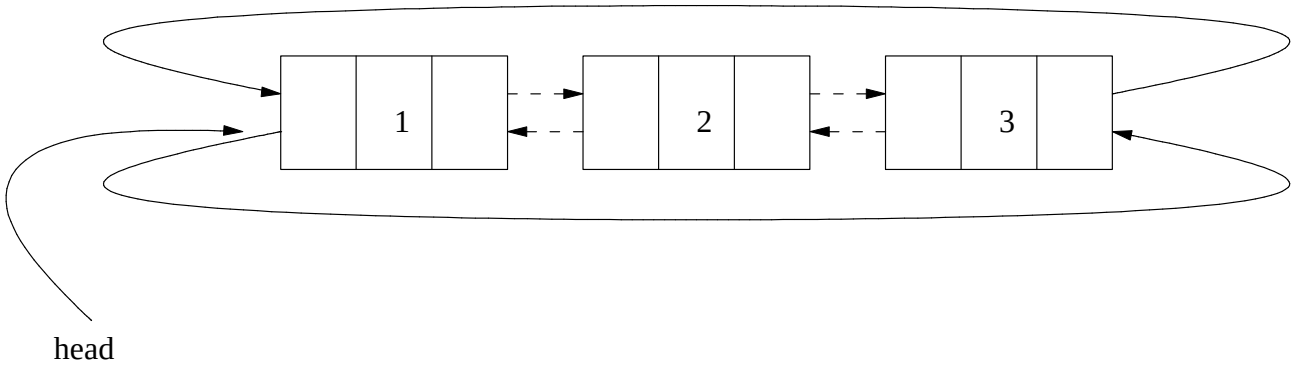
# Double linked list implementation in C++

- recall our single list implementation earlier in the series
- many times in Computer Science we need to maintain dynamically adjustable lists
- consider if we had two lists and we had a number of elements on each
  - and your application needed to frequently move elements from one list to another and visa versa
- this could prove costly if implemented with a single linked list

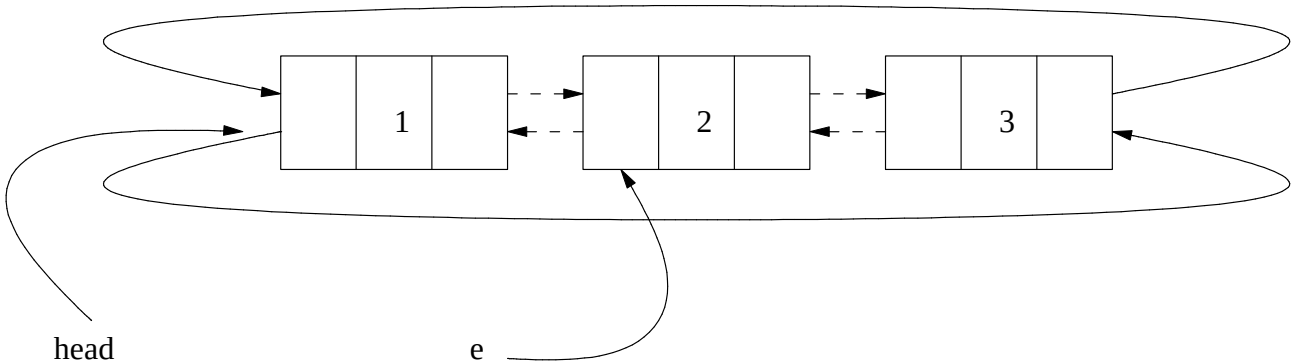
# Double linked list implementation in C++

- especially if you know the element which needs to be removed
  - the single list implementation needs to always scan from the head down to the known element
  - there after it can remove it

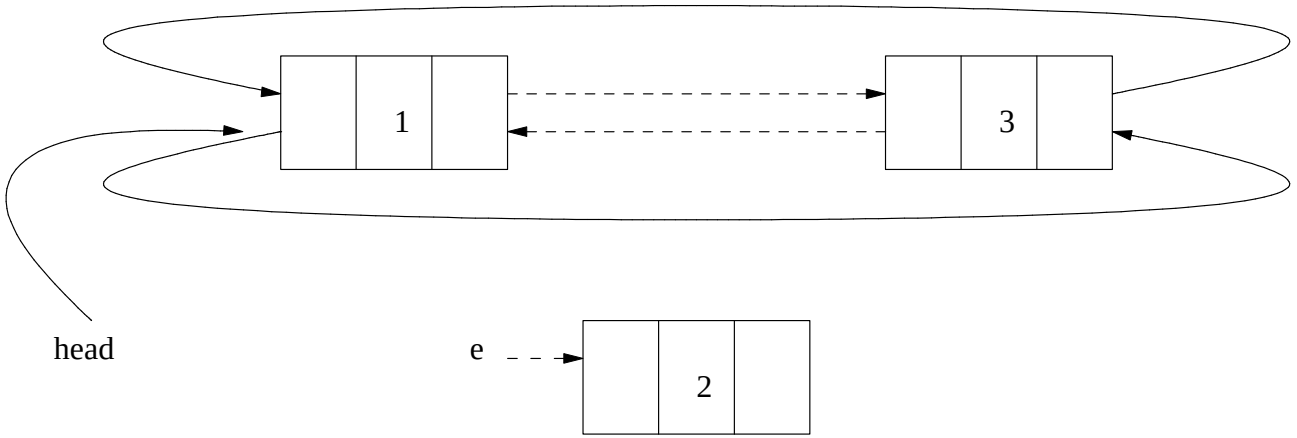
# Double linked list data structure



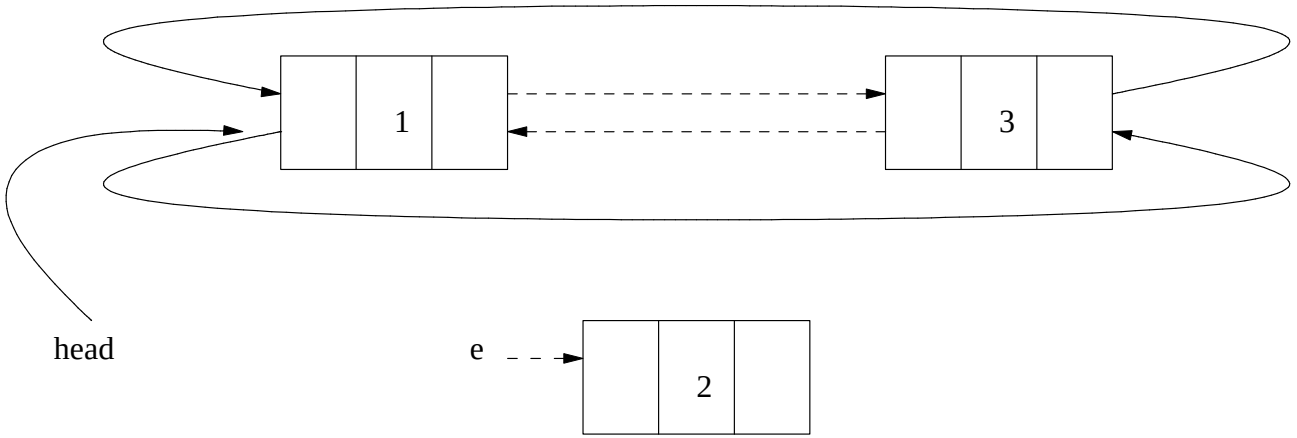
# Removing an element



# Removing an element



# Pointer reminder




- which boxes are affected by the following operations?
  - `e->right = 100`
  - `head->right = 100`
  - `head->right->right = 100`
  - `head->left->right = 100`
  - `head->left->left = 100`

# Advantages of double linked list

- we notice that we can remove an element without, knowing where on the list it resides
- can easily find the tail element (`head->left`)
- can easily add an element to the end without having to traverse chain using `next` as in our `slist` implementation

# Disadvantages of double linked list

- slightly more complexity
- an additional pointer per element (`right` and `left` rather than `next`)
- slightly harder to write the code for list iteration as it is now circular
  - no easy `null` pointer to signify end of list

# Definition of the class

**c++/lists/double-list/int/v1/dlist.h**

```
#if !defined(DLISTH)
#  define DLISTH

#include <iostream>

class element
{
 public:
  element *left;
  element *right;
  int      data;
};
```

# Definition of the class

**c++/lists/double-list/int/v1/dlist.h**

```
class dlist
{
 private:
  element *head_element;
  element *duplicate_elements (element *e);
  void     delete_elements (void);
  dlist    cons (element *e);
  friend std::ostream& operator<< (std::ostream& os, const dlist& l);

 public:
  dlist (void);
  ~dlist (void);
  dlist (const dlist &from);
  dlist& operator= (const dlist &from);
```

# Definition of the class

**c++/lists/double-list/int/v1/dlist.h**

```
dlist empty (void);
bool  is_empty (void);
dlist cons (int i);
int   head (void);
dlist tail (void);
dlist cons (dlist l);
dlist reverse (void);
int length (void);

void add (element **head, element *e);
void delete_element (element *e);
element *sub (element *e);
```

# Definition of the class

**c++/lists/double-list/int/v1/dlist.h**

```
  dlist append (int i);
  dlist cut (int low, int high);
  dlist slice (int low, int high);
  dlist ordered_insert (int i);
};
#endif
```

# dlist constructor

**c++/lists/double-list/int/v1/dlist.cc**

```
/*
 *  dlist - constructor, builds an empty list.
 *          pre-condition:    none.
 *          post-condition:   list is created and is empty.
 */

dlist::dlist (void)
  : head_element(0)
{
}
```

# How do we remove an element from the list?

**c++/lists/double-list/int/v1/dlist.cc**

```
/*
 *  sub - remove, e, from the list.
 *        Pre-condition : e is on the list.
 *        Post-condition: e is removed and returned.
 */

element *dlist::sub (element *e)
{
  if ((e->right == head_element) && (e == head_element))
    /*
     *  e is the only element on the list.
     */
    head_element = 0;
  else
```

# How do we remove an element from the list?

**c++/lists/double-list/int/v1/dlist.cc**

```
    {
      if (head_element == e)
        /*
         *  be prepared to move head_element on one element
         */
        head_element = head_element->right;

      /*
       *  now unhook, e, from our double linked list
       */
      e->left->right = e->right;
      e->right->left = e->left;
    }
  return e;
}
```

# Addition to the list

```
/*
 *  add - add, e, to the end of list.
 *        pre-condition :  e is not on a list.
 *        post-condition:  e is appended to the end of the list.
 */

void dlist::add (element *e)
{
  if (head_element == 0)
    {
      head_element = e; // head is empty therefore make
      e->left = e;      // e the only entry on this
      e->right = e;     // list.
    }
```

# Addition to the list

```
  else
    {
      e->right = head_element;  // add e to the end of list
      e->left = head_element->left;  // copy the current end to e
      head_element->left->right = e;  // add e to the end of the last
      head_element->left = e;  // alter the left of head to point to e
    }
}
```

- however we find in the `dlist` implementation we need to create new lists of elements which must not be refered to by `head_element`
  - for example `duplicate_elements`

# add

**c++/lists/double-list/int/v1/dlist.cc**

```
/*
 *  add - add, e, to the end of list, defined by (*head).
 *        pre-condition :  (*head) points to a list.
 *                        e is not on a list.
 *        post-condition:  e is appended to the end of head.
 */

void dlist::add (element **head, element *e)
{
  if (*head == 0)
    {
      *head = e;          // head is empty therefore make
      e->left = e;        // e the only entry on this
      e->right = e;       // list.
    }
```

# add

- c++/lists/double-list/int/v1/dlist.cc

```
  else
    {
      e->right = *head;  // add e to the end of list
      e->left = (*head)->left;  // copy the current end to e
      (*head)->left->right = e;  // add e to the end of the last
      (*head)->left = e;  // alter the left of head to point to e
    }
}
```

- `(*head)` can be thought of as `head_element`
  - `add` is implemented using `(*head)` so that other methods can build up lists independent of `head_element`

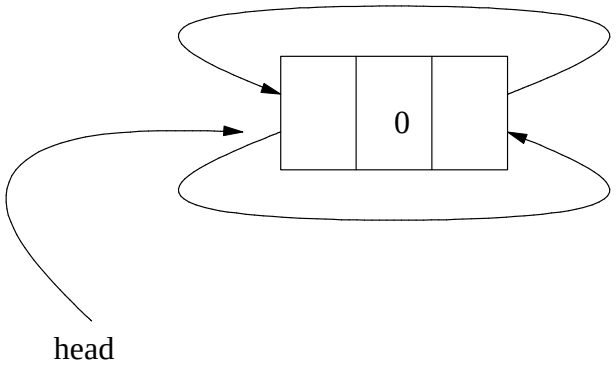
# Observations

- notice that in both `add` and `sub` we need to test whether
  `head_element == 0`

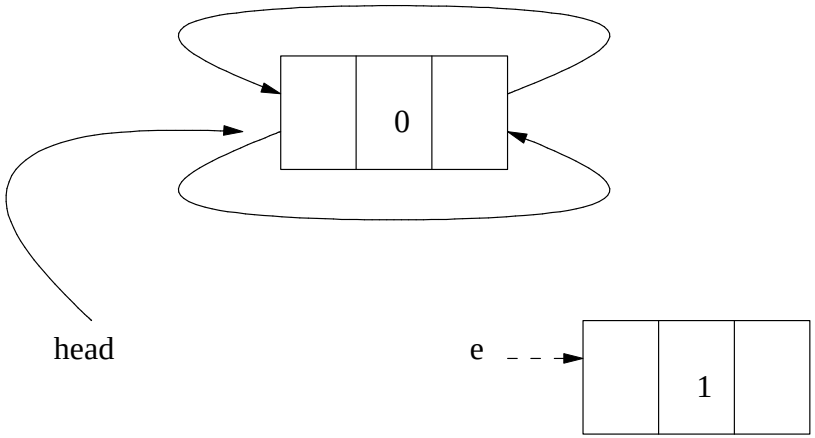
- we could remove these conditional tests if we ensure we always have at least one element on the list
  - sentinel element (see D. Knuth, Fundamental Algorithms Volume 1, 2nd Edition, 1973, P.278)

# Double linked list with sentinel value

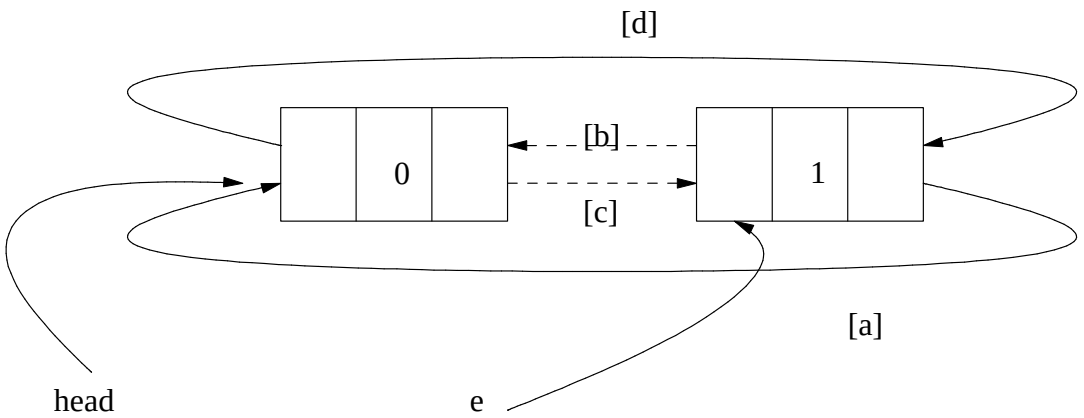
- constructed as

- 

# Adding to our initialised list



# Adding to our initialised list




**c++/lists/double-list/int/v2/dlist.cc**

```
[a] e->right = head;  // add e to the end of list
[b] e->left = head->left;  // copy the current end to e
[c] head->left->right = e;  // add e to the end of the last
[d] head->left = e;  // alter the left of head to point to e
```

# Subtracting from our list

**c++/lists/double-list/int/v2/dlist.cc**

```
/*
 *  sub - remove, e, from the list.
 *        Pre-condition : e is on the list.
 *        Post-condition: e is removed and returned.
 */

element *dlist::sub (element *e)
{
  assert (! (head_element == e));
  assert (! ((e->right == head_element) && (e == head_element)));
  /*
   *  now unhook, e, from our double linked list
   */
  e->left->right = e->right;
  e->right->left = e->left;
  return e;
}
```

# Subtracting from our list

- notice that ignoring any asserts we no longer need to check whether `head_element == 0`

# Performance gains

- this can be a large win in time critical code as we avoid cache misses
  - no branching necessary

- often used in microkernels and real-time systems
  - where we need to move a process from the running list to a blocked list
  - and visa-versa

# Tutorial

- run the `sub` code by hand redrawing the diagram after each pointer change

- run the `add` code by hand redrawing the diagram after each pointer change

# Tutorial

- examine the implementation of the method: `dlist::cons (element *e)`
  - in file `c++/lists/double-list/int/v2/dlist.cc`

- draw two example lists of two element each
  - let one list exist in `this` and another defined by `e`
  - now execute the method by hand
  - redraw your diagram every time a pointer is changed

# Tutorial

- implement the following methods:

# Tutorial

```
/*
 *  slice - return a copy of the list between low..high-1.
 *          pre-condition:  an initialised non empty list.
 *          post-condition: the original list is unaltered.
 *                          A new list is returned which is
 *                          a copy of elements, low..high-1
 *                          Negative values of low and high
 *                          index from the right (aka Python).
 */

dlist dlist::slice (int low, int high);
```

# Tutorial

```
/*
 *  cut - return a region of the list.
 *        pre-condition:  an initialised non empty list.
 *        post-condition: elements low..high are removed from
 *                        the original list.
 *                        A new list is returned which contains
 *                        the elements, low..high-1
 *                        Negative values of low and high
 *                        index from the right (aka Python).
 */

dlist dlist::cut (int low, int high);
```