

[Help](#)

```
/*
 * Written by David Pommier <david.pommier@gmail.com>
 * INRIA 2009
 */

/**
 * allocates a Node - use contains copy constructor.
 * @param Val a CONTAIN pointer
 * @return a pointeur to TYPE(PremiaNode)
 */
NODE *FUNCTION_NODE(premia_, create)(const CONTAIN *Val)
{
    NODE *N;
    if ((N = malloc(sizeof(NODE))) == NULL) return NULL;
    N->previous = NULL;
    N->next = NULL;
    N->obj = FUNCTION_CONTAIN(premia_, copy)(Val);
    return N;
}

/**
 * allocates a Node - use contains copy constructor.
 * @param key a KEY
 * @param val a VALUE
 * @return a pointeur to TYPE(PremiaNode)
 */
NODE *FUNCTION_NODE(premia_, create_from_key_val)(const KEY key, const VALUE val)
{
    NODE *N;
    if ((N = malloc(sizeof(NODE))) == NULL) return NULL;
    N->previous = NULL;
    N->next = NULL;
    N->obj = FUNCTION_CONTAIN(premia_, create)(key, val);
    return N;
}

/**
 * free a Node
```

```

    * @param node address of a NODE
    */
void FUNCTION_NODE(premia_, free)(NODE **node)
{
    if (*node != NULL)
    {
        FUNCTION_CONTAIN(premia_, free)(&((*node)->obj));
        free(*node);
        *node = NULL;
    }
}

/**
 * Do a shift of current,
 * n times current=current->next if n>0
 * n times current=current->previous if n<0
 *
 * @param current address of a NODE
 * @param n a int
 */
void FUNCTION_NODE(premia_, shift)(NODE **current, int n)
{
    int m = n;
    if (m < 0)
        while (m != 0)
        {
            *current = (*current)->previous;
            m++;
        }
    else
        while (m != 0)
        {
            *current = (*current)->next;
            m--;
        }
}

/**
 * Do a search Val and return current,
 * if Val.Key is in List, then current is a pointer on this node
 * else current in next or before node, and result indicate the position

```

```

*
* @param current address of a NODE
* @param Val a pointer on CONTAIN
* @result a int,
* 0 if in the list,
*-1, not in the list and current is the next node,
* 1 if not in the list and current is the previous node
*/
int FUNCTION_NODE(premia_, search)(NODE **current, const CONTAIN *Val)
{
    while (((*current)->next != NULL) && (FUNCTION_CONTAIN(premia_, less)((*current)->obj, Val) != 0))
        (*current) = (*current)->next;
    if ((*current)->next == NULL)
    {
        int sg = FUNCTION_CONTAIN(premia_, less)((*current)->obj, Val);
        if (sg == FUNCTION_CONTAIN(premia_, less)(Val, (*current)->obj))
            return 0;
        // it is equal
        return (sg > 0) ? 1 : -1;
        // Add a end of list (add by next)
    }
    if (FUNCTION_CONTAIN(premia_, less)(Val, (*current)->obj) == 0)
        return 0; // it is equal
    // Add just before current, (add by previous)
    return -1;
}

/**
* creates a new TYPE(PremiaSortList).
*
* @return a TYPE(PremiaSortList) pointer
*/
TYPE(PremiaSortList) *FUNCTION(premia_, create)()
{
    TYPE(PremiaSortList) * List;
    if ((List = malloc(sizeof(TYPE(PremiaSortList)))) == NULL) return NULL;
    List->size = 0;
    List->first = NULL;
    List->last = NULL;
    List->current = NULL;
}

```

```

    return List;
}

/**
 * creates a new TYPE(PremiaSortList) pointer.
 *
 * @param List a TYPE(PremiaSortList) to copy
 * @return a TYPE(PremiaSortList) pointer
 */
TYPE(PremiaSortList) *FUNCTION(premia_, clone)(TYPE(PremiaSortList) * List)
{
    TYPE(PremiaSortList) * List2;
    if ((List2 = malloc(sizeof(TYPE(PremiaSortList)))) == NULL) return NULL;
    List2->size = List->size;
    List2->first = List->first;
    List2->last = List->last;
    List2->current = List->current;
    return List2;
}

/**
 * free a TYPE(PremiaSortList) pointer and set the data pointer to
 * NULL
 *
 * @param List address of the pointer to free
 */
void FUNCTION(premia_, free)(TYPE(PremiaSortList) **List)
{
    NODE *current;
    if (*List == NULL) return;

    current = (*List)->first;
    if (current != NULL)
    {
        while (current->next != NULL)
        {
            current = current->next;
            FUNCTION_NODE(premia_, free>(&(current->previous));
        }
        FUNCTION_NODE(premia_, free>(&(current)));
    }
}

```

```

        free(*List);
        *List = NULL;
    }
}

```

```

static int FUNCTION(premia_, search_dicho_recc)(TYPE(PremiaSortList) * List, NOD
{
    int before;
    if (List->size == 2)
    {
        *current = List->first;
        before = FUNCTION_CONTAIN(premia_, less)(Val, (*current)->obj);
        if (before)
            return -1;
        if (before == FUNCTION_CONTAIN(premia_, less)((*current)->obj, Val))
            return 0;
        else
        {
            *current = List->last;
            before = FUNCTION_CONTAIN(premia_, less)(Val, (*current)->obj);
            if (before)
                return -1;
            if (before == FUNCTION_CONTAIN(premia_, less)((*current)->obj, Val))
                return 0;
            return 1;
        }
    }
    if (List->size == 1)
    {
        *current = List->first;
        before = FUNCTION_CONTAIN(premia_, less)((*current)->obj, Val);
        if (before == FUNCTION_CONTAIN(premia_, less)(Val, (*current)->obj))
            return 0;
        return (before) ? 1 : -1;
    }
    *current = List->first;
    FUNCTION_NODE(premia_, shift)(current, (List->size) / 2);
    before = (FUNCTION_CONTAIN(premia_, less)(Val, (*current)->obj) == 1) ? 1 : 0;
    if (FUNCTION_CONTAIN(premia_, less)((*current)->obj, Val) == before)

```

```

    return 0;
//is equal
if (before)
{
    List->last = (*current)->previous;
    List->size /= 2;
    // Size=(size-1)/2 for even it's n/2 - 1 and n/2 for odd
    // -1 come from we exclue current for the next reccursive step
}
else
{
    List->first = (*current)->next;
    List->size -= 1;
    List->size /= 2;
    // Size=(size+1)/2 for even it's (n)/2 +1 - 1 and n/2 +1 for odd
    // -1 come from we exclue current for the next reccursive step
}
// Search on right sub list
return FUNCTION(premia_, search_dicho_recc)(List, current, Val);
}

```

```

static int FUNCTION(premia_, search)(TYPE(PremiaSortList) * List, NODE **current)
{
    *current = List->first;
    return FUNCTION_NODE(premia_, search)(current, Val);
}

```

```

static int FUNCTION(premia_, search_dicho)(TYPE(PremiaSortList) * List, NODE **current)
{
    if (FUNCTION_CONTAIN(premia_, less)(Val, List->first->obj) == 1)
        // Add to left (before first)
        {
            *current = List->first;
            return -1;
        }
    if (FUNCTION_CONTAIN(premia_, less)(List->last->obj, Val) == 1)
        // Add to right (after last)
        {
            *current = List->last;
            return 1;
        }
}

```

```

{
    TYPE(PremiaSortList) * L = FUNCTION(premia_, clone)(List);
    int where_add;
    where_add = FUNCTION(premia_, search_dicho_recc)(L, current, Val);
    free(L);
    return where_add;
}
}

static void FUNCTION(premia_, insert)(TYPE(PremiaSortList) * List, const CONTAIN
                                     int (*search)(TYPE(PremiaSortList) *, NODE
                                     void (*operator)(CONTAIN *, const CONTAIN

{

    if (List->size == 0)
    {
        NODE *current = FUNCTION_NODE(premia_, create)(Val);
        List->first = current;
        List->last = current;
        List->size++;
    }
    else
    {
        NODE *current;
        int where_add = search(List, &current, Val);
        if (where_add == 1)
            //add in last position
            {
                if (current == List->last)
                {
                    current->next = FUNCTION_NODE(premia_, create)(Val);
                    current->next->previous = current;
                    List->last = current->next;
                }
                else
                {
                    current->next->previous = FUNCTION_NODE(premia_, create)(Val);
                    current->next->previous->next = current->next;
                    current->next = current->next->previous;
                    current->next->previous = current;
                }
            }
    }
}

```

```

        List->size++;
    }
    else if (where_add == -1)
    {
        if (current == List->first)
        {
            List->first = FUNCTION_NODE(premia_, create)(Val);
            current->previous = List->first;
            current->previous->next = current;
        }
        else
        {
            current->previous->next = FUNCTION_NODE(premia_, create)(Val);
            current->previous->next->previous = current->previous;
            current->previous = current->previous->next;
            current->previous->next = current;
        }
        List->size++;
    }
    else
        operator(current->obj, Val);
}

static int FUNCTION(premia_, find_withf)(TYPE(PremiaSortList) * List, NODE **current,
int (*search)(TYPE(PremiaSortList) *, NODE **, const CONTAIN *))
{
    if (List->size == 0)
    {
        NODE *first_node = FUNCTION_NODE(premia_, create_from_key_val)(key, val);
        List->first = first_node;
        List->last = first_node;
        (*current) = first_node;
        List->size++;
        return 1;
    }
    else
    {
        CONTAIN *Val;
        int where_add;
        Val = FUNCTION_CONTAIN(premia_, clone)(key, val);

```



```

where_add = search(List, current, Val);
FUNCTION_CONTAIN(premia_, free)(&Val);
if (where_add == 1)
    //add in last position
    {
        if ((*current) == List->last)
        {
            (*current)->next = FUNCTION_NODE(premia_, create_from_key_val)(key_val);
            (*current)->next->previous = (*current);
            List->last = (*current)->next;
            (*current) = List->last;
        }
        else
        {
            (*current)->next->previous = FUNCTION_NODE(premia_, create_from_key_val)(key_val);
            (*current)->next->previous->next = (*current)->next;
            (*current)->next = (*current)->next->previous;
            (*current)->next->previous = (*current);
            (*current) = (*current)->next;
        }
        List->size++;
        return 1;
    }
else if (where_add == -1)
{
    if ((*current) == List->first)
    {
        (*current)->previous = FUNCTION_NODE(premia_, create_from_key_val)(key_val);
        (*current)->previous->next = (*current);
        List->first = (*current)->previous;
        (*current) = List->first;
    }
    else
    {
        (*current)->previous->next = FUNCTION_NODE(premia_, create_from_key_val)(key_val);
        (*current)->previous->next->previous = (*current)->previous;
        (*current)->previous = (*current)->previous->next;
        (*current)->previous->next = (*current);
        (*current) = (*current)->previous;
    }
    List->size++;
}

```

```

        return 1;
    }
    return 0; // Not added
}
}

/**
 * Find a Contains to a TYPE(PremiaSortList)
 * So return pointer on element of a list.
 * if not in the list adding before
 *
 * @param List a(constant) TYPE(PremiaSortList) ptr.
 * @param current address of a pointer on NODE .
 * @param key a KEY.
 * @param val a VALUE.
 * @result return 1 if Val is added to List 0 either
 */
int FUNCTION(premia_, find)(TYPE(PremiaSortList) * List, NODE **current, KEY key)
{
    return FUNCTION(premia_, find_withf)(List, current, key, val, FUNCTION(premia_,
}

/**
 * Find a Contains to a TYPE(PremiaSortList)
 * So return pointer on element of a list.
 * if not in the list adding before
 *
 * @param List a(constant) TYPE(PremiaSortList) ptr.
 * @param current address of a pointer on NODE .
 * @param key a KEY.
 * @param val a VALUE.
 * @result return 1 if Val is added to List 0 either
 */
int FUNCTION(premia_, find_dicho)(TYPE(PremiaSortList) * List, NODE **current, K
{
    return FUNCTION(premia_, find_withf)(List, current, key, val, FUNCTION(premia_
}

/**
 * Add a Contains to a TYPE(PremiaSortList)

```

```

* So add to a list if not in te list else add value,
* see FUNCTION(premia_contains,add) for action on Contians.
*
* @param List a(constant) TYPE(PremiaSortList) ptr.
* @param Val a CONTAIN.
*/
void FUNCTION(premia_, add)(TYPE(PremiaSortList) * List, const CONTAIN *Val)
{
    FUNCTION(premia_, insert)(List, Val, FUNCTION(premia_, search), FUNCTION_CONTA
}

/**
* Add a Contains to a TYPE(PremiaSortList)
* So add to a list if not in te list else add value,
* see FUNCTION(premia_contains,add) for action on Contians.
* use dichotomic search to fast insertion operation
*
* @param List a(constant) TYPE(PremiaSortList) ptr.
* @param Val a CONTAIN.
*/void FUNCTION(premia_, add_dicho)(TYPE(PremiaSortList) * List, const CONTAIN
{
    FUNCTION(premia_, insert)(List, Val, FUNCTION(premia_, search_dicho), FUNCTION
}

/**
* prints a TYPE(PremiaSortList) in file fic.
*
* @param List a(constant) TYPE(PremiaSortList) ptr.
* @param fic a file descriptor.
*/
void FUNCTION(premia_, fprint)(FILE *fic, const TYPE(PremiaSortList) * List)
{
    NODE(*current) = List->first;
    while (current != List->last)
    {
        FUNCTION_CONTAIN(premia_, fprint)(fic, current->obj);
        current = current->next;
    }
    FUNCTION_CONTAIN(premia_, fprint)(fic, List->last->obj);

```

```

/*
// double loop test previous
while(current!=List->first)
{
FUNCTION(premia_contains,fprint(fic,current->obj);
current=current->previous;
}
FUNCTION(premia_contains,fprint(fic,List->first->obj);
*/
fprintf(fic, "\ n");
}

/**
 * prints a TYPE(PremiaSortList).
 *
 * @param List a(constant) TYPE(PremiaSortList) ptr.
 */
void FUNCTION(premia_, print)(const TYPE(PremiaSortList) * List)
{
    if (List->size > 0)
        FUNCTION(premia_, fprint)(stdout, List);
}

```