llist.h

```
#ifndef LLIST_H_INCLUDE
#define LLIST_H_INCLUDE

#include <stdlib.h> // size_t
#include <stdint.h>
#include <stdbool.h>

typedef struct linkedlist_t LinkedList;
typedef struct linkednode_t LinkedNode;

/**
 * Iterator type for iterating though a linked list in linear time
 */
typedef struct linkediterator_t
{
  LinkedList* list;
  LinkedNode* current;
} LinkedIterator;

/**
 * Constructs and returns a new, empty, doubly linked list
 * @return linked list
 */
LinkedList* ll_new();

/**
 * Returns the number of elemnts in the linked list
 * @param  ll The linked list to find the number of elements for
 * @return    The size
 */
size_t ll_size(const LinkedList* const ll);
/**
 * Returns true if the linked list is empty (size == 0)
 * @param  ll The linked list to test
 * @return    True if the linked list is empty
 */
```

```
bool ll_empty(const LinkedList* const ll);

/**
 * Returns the data at the front of the list
 * @param ll The linked list to get the first element from
 */
void* ll_front(const LinkedList* const ll);
/**
 * Returns the data at the back of the list
 * @param ll The linked list to get the last element from
 */
void* ll_back(const LinkedList* const ll);

/**
 * Push a piece of data onto the front of the list
 * @param ll The linked list to push onto
 * @param p  The data to psuh
 */
void ll_push_front(LinkedList* const ll, void* p);
/**
 * Push a piece of data onto the back of the list
 * @param ll The linked list to push onto
 * @param p  The data to push
 */
void ll_push_back(LinkedList* const ll, void* p);

/**
 * Pop a piece of data off the front of the list
 * @param ll The list to pop from
 */
void ll_pop_front(LinkedList* const ll);
/**
 * Pop a piece of data off the back of the list
 * @param ll The list to pop from
 */
void ll_pop_back(LinkedList* const ll);

/**
 * Inserts the data at p before the position given by it
 * @param it Insert before
 * @param p  Data to insert
 */
void ll_insert(LinkedIterator* const it, void* p);

/**
 * Removes the element at the given iterator from the list.
```

```
 * The suppled iterator is invalidated, the data is not freed and must be
 * handled separately
 * @param it The iterator to remove
 */
void* ll_erase(LinkedIterator* const it);

/**
 * Retrieve a piece of data in the list at a specific index.
 * Note that this is a O(n) operation, and therefore not suitable for
 * iterating the linked list.
 * @param ll The linked list to get data from
 * @param n  The index
 */
LinkedIterator ll_at(const LinkedList* const ll, const size_t n);

/**
 * Sort the linked list using a bubble sort
 * @param ll The linked list to sort
 * @param comparison A function pointer for comparing elements
 */
void ll_bsort(LinkedList* const ll, int32_t (*comparison)
                            (const void* const a, const void* const b));

/**
 * Empties a linked list, freeing all the nodes and their data
 * @param ll The linked list to clear
 */
void ll_clear(LinkedList* ll);
/**
 * Destructs a list, freeing all data, nodes and the linked list itself
 * Calling this invalidates the pointer, as the memory is freed
 * @param ll The linked list to delete
 */
void ll_delete(LinkedList* ll);
/**
 * Destructs a list, but does not free the data unlike ll_delete
 * If a pointer for each piece of data is not kept, it is leaked.
 * @param ll The link list to purge
 */
void ll_purge(LinkedList* ll);


/**
 * Begins iteration at the front of the list, can be called
 * as many times as you like on a single iterator or list to restart iteration
 * @param it Iterator struct to control iteration
```

```
 * @param ll List to iterate over
 */
LinkedIterator ll_it_begin(LinkedList* ll);
/**
 * Begins iteration at the back of the list, can be called
 * as many times as you like on a single iterator or list to restart iteration
 * @param it Iterator struct to control iteration
 * @param ll List to iterate over
 */
LinkedIterator ll_it_rbegin(LinkedList* ll);
/**
 * Steps iteration forwards though a list. Returns null at the end of the list
 * @param it Iterator to step forwards
 * @return Data of the next node, or NULL at the end of the list
 */
void* ll_it_next(LinkedIterator* const it);
/**
 * Steps iteration backwards though a list. Returns null at the end of the list
 * @param it Iterator to step backwards
 * @return Data of the next node, or NULL at the end of the list
 */
void* ll_it_rnext(LinkedIterator* const it);

/**
 * Returns true if this iterator points to a valid piece of data
 * @param  it The iterator to test
 * @return    True if the data is valid
 */
bool ll_it_valid(const LinkedIterator* const it);

/**
 * Returns the data at a specific element.
 * Returns NULL if the iterator is invalid
 * @param it the iterator to get the data for
 */
void* ll_it_data(const LinkedIterator* const it);
#endif
```