Readme.md

Quick Sort with AList

The code is in `rsort_and_qsort` folder.

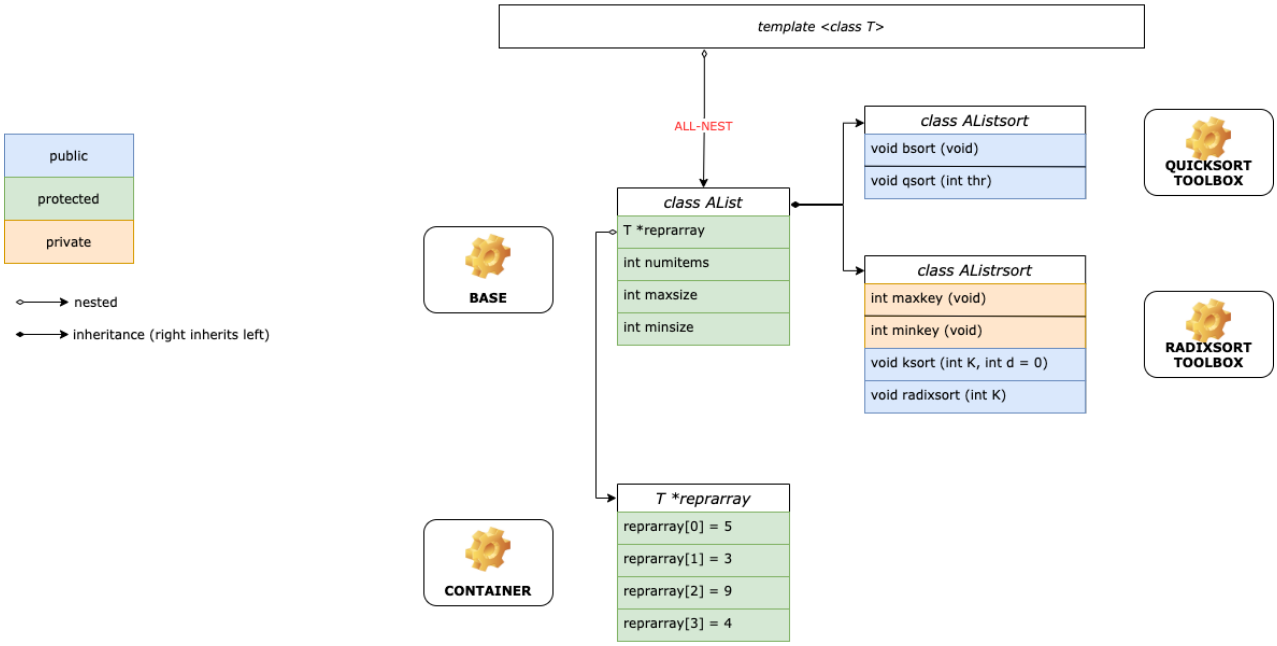
The principle is pretty similar to the merge sort, because they both use **divide & conquer** algorithm. It uses an embedded `bsort()` to perform.

Inner data structure

Radix Sort & Quick Sort Inner Data Structures

@Jack Get Bugs!

Class Exploration



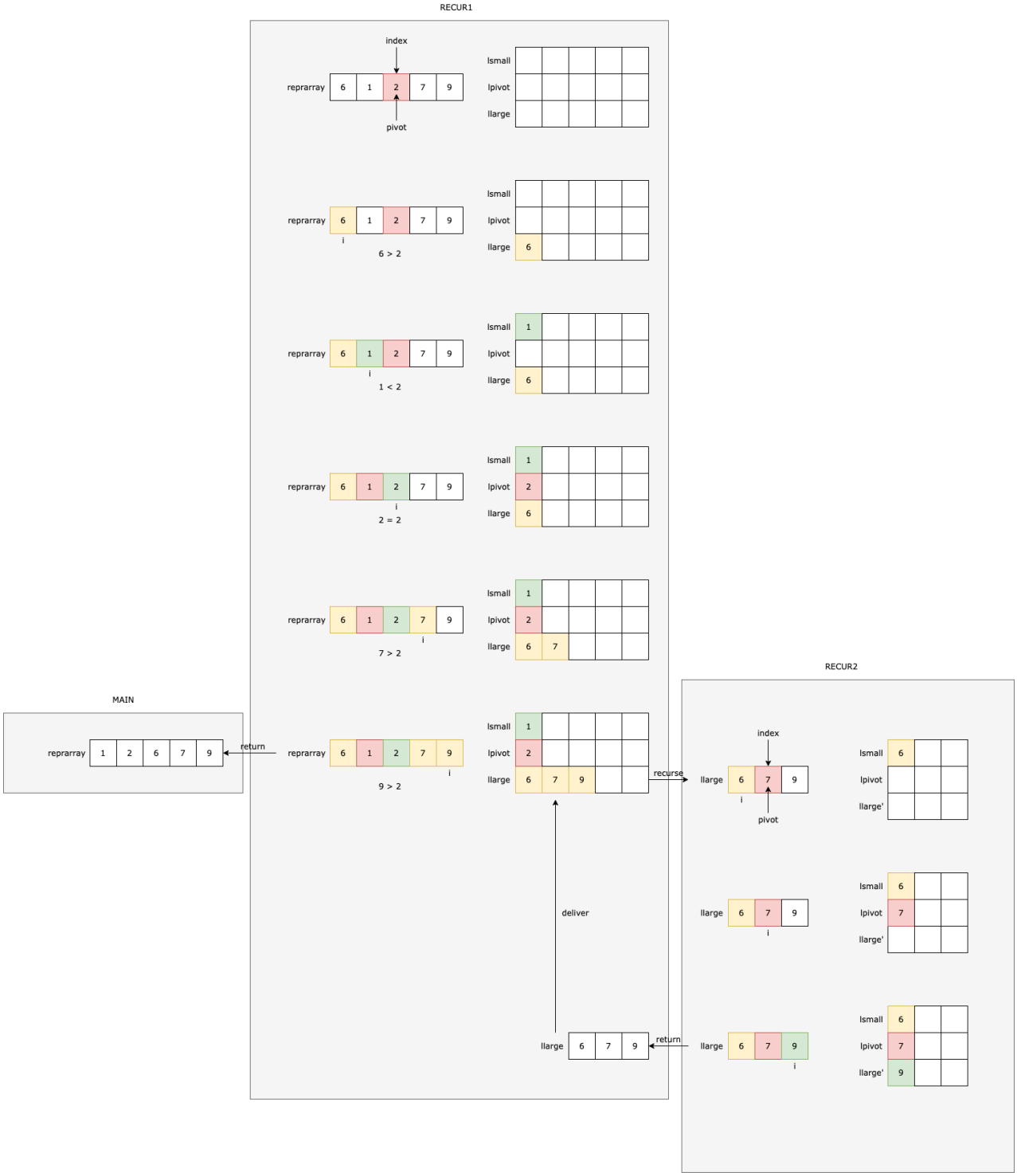
Bubble Sort

This is too easy so no graphs are provided.

Quicksort

Quick Sort Using AList<T>

@Jack Get Bugs!

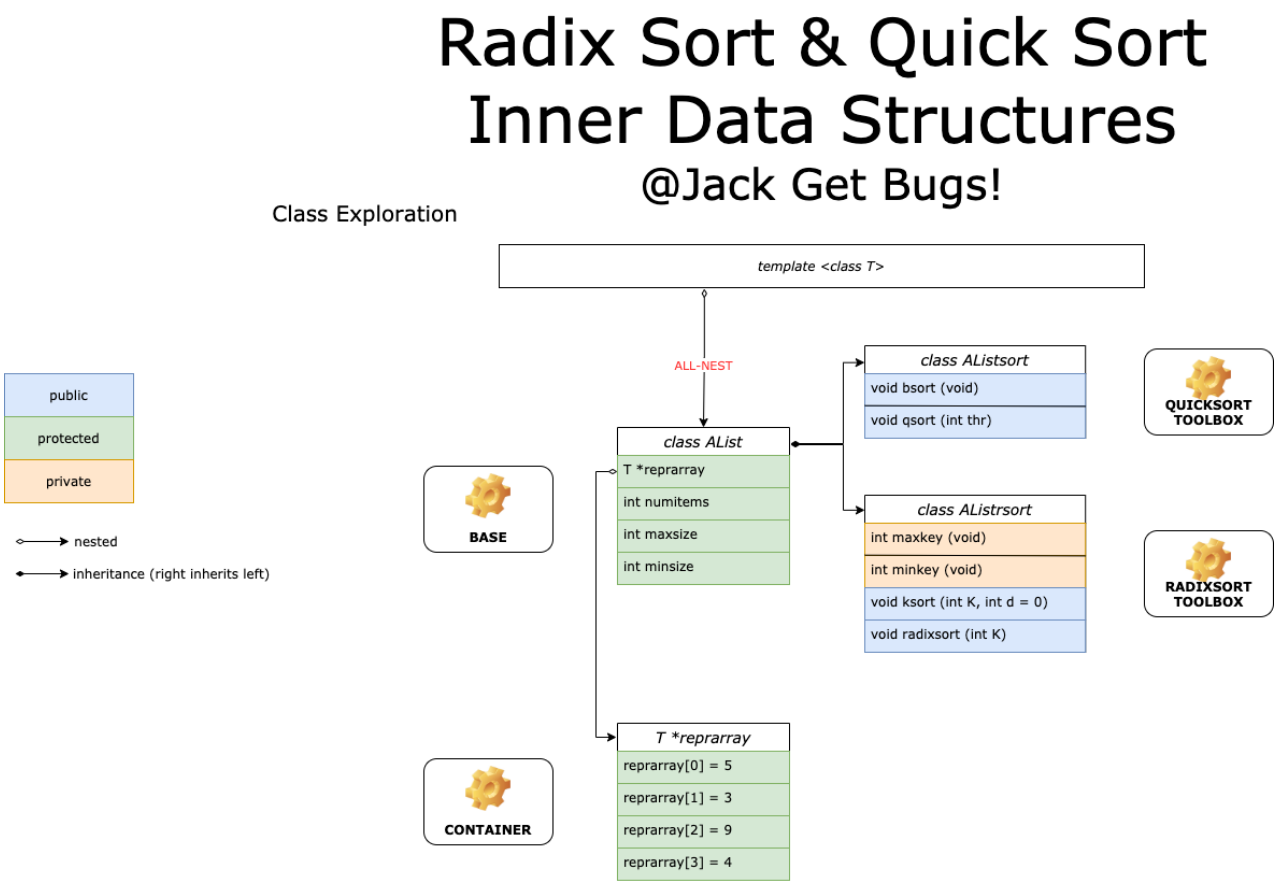


Radix Sort with AList

The code is in `rsort_and_qsort` folder.

Radix Sort is all about playing cards. It uses an embedded `ksort()` to perform.

Inner Data Structures



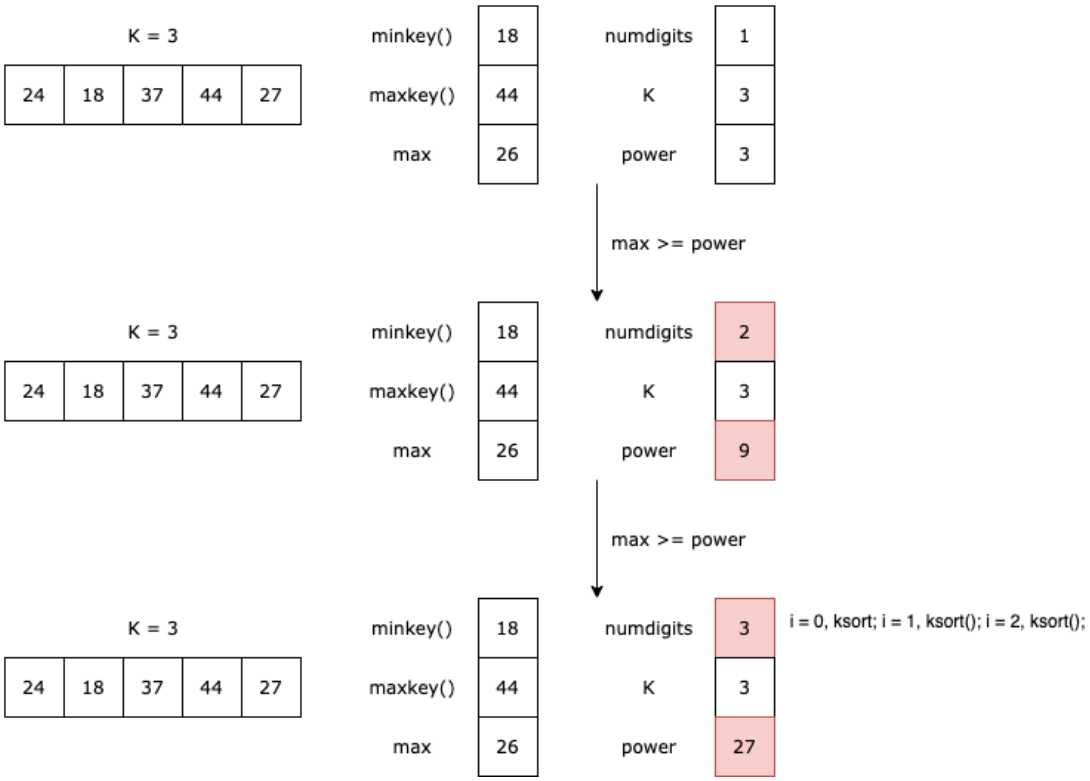
Radix Sort

Compared to selection sort, heap sort and merge sort, radix sort is **not** a sorting algorithm based on comparison. Note that any sorting algorithm based on comparison requires a complexity of at least $O(n \log n)$. Thus, radix sort & bucket sort is one way to break this through - they don't necessarily need a lower bound.

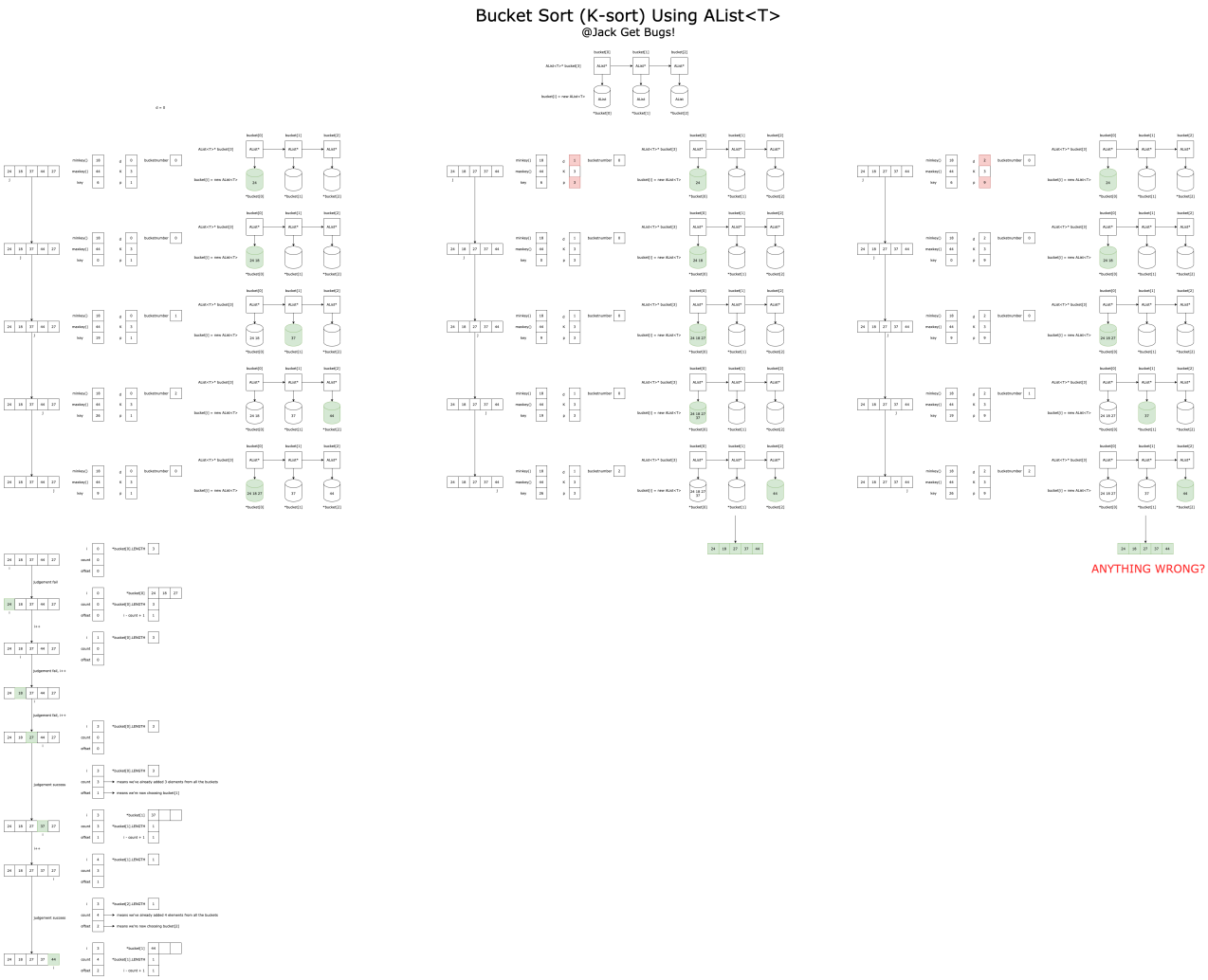
However, there seems to be some glitches (either by me or the writer), so I'd suggest for a more authoritative explanation for this part (incl. Bucket sort & Radix sort).

Radix Sort Using AList<T>

@Jack Get Bugs!



Bucket Sort (K-sort)



Merge Sort with DList

The code is in `mergeSort` folder.

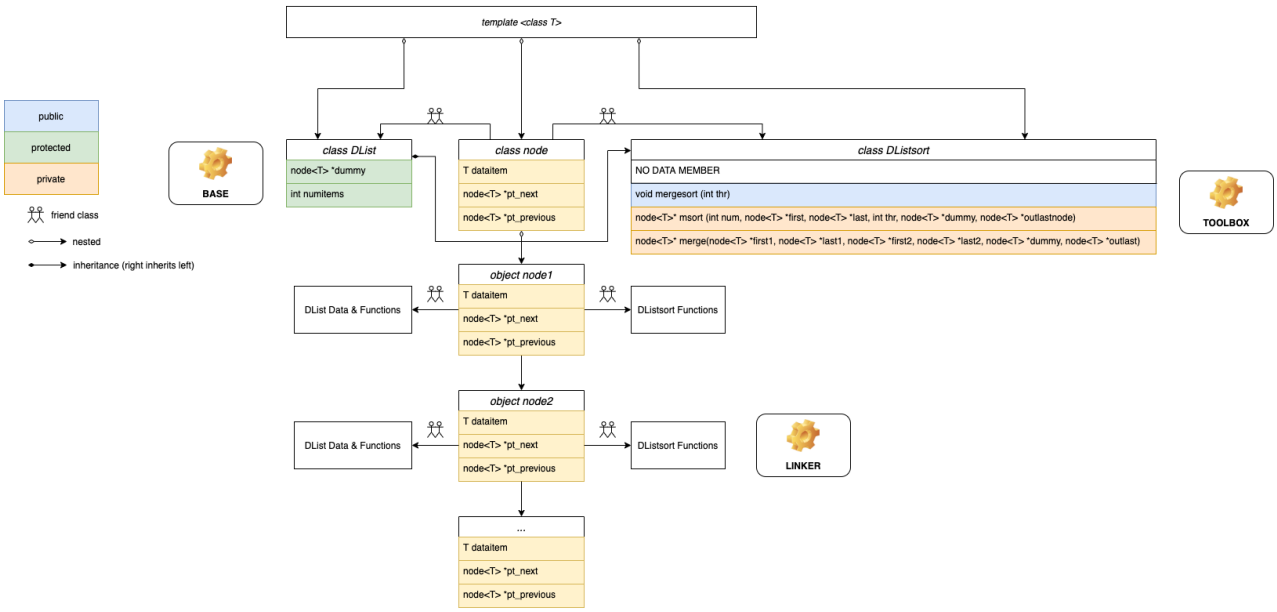
Merge Sort is one of the most familiar sorting algorithms to us. It uses an embedded `ssort()` to perform.

Inner Data Structures

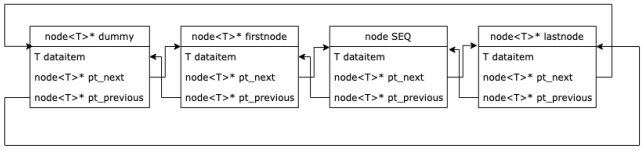
Inner Data Structures

@Jack Get Bugs!

Class Exploration



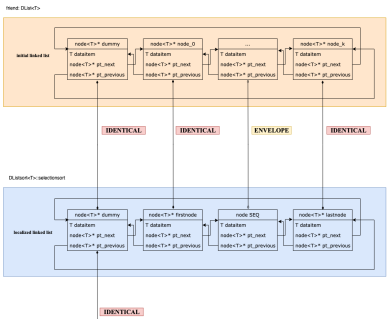
Data Structure Exploration



Selection Sort

Selection Sort Using DList<T>

@Jack Get Bugs!

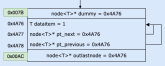



```
// localize the numitems & dummy.
// the usage of dummy is illustrated in the graph I drew.
int numitems = DList<T>::numitems;
node<T> * dummy;
dummy = DList<T>::dummy;

// when the total number of elements in the list is smaller or equal to 1,
//  then return immediately.
if (numitems <= 1)
    return;

// define the firstnode and last node here.
node<T> *firstnode;
firstnode = (*dummy).getnext();
node<T> *lastnode;
lastnode = (*dummy).getprevious();
```

DList<int>::selectionsort

STATE 1

new linked list




```
// initialize outlastnode.
node<T> *outlastnode;
outlastnode = dummy;

// isolate dummy so that it forms a new linked list.
// NOTE: dummy act as the head of the new linked list.
(*outlastnode).setnext(dummy);
(*outlastnode).setprevious(dummy);
```

```
template<class T> void node<T>::setprevious(node<T> *pt)
{
    pt_previous = pt;
    return;
}
```

STATE 2


```
// add minnode between outlastnode & dummy.
add(minnode, outlastnode, dummy);
```

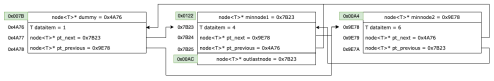
```
/* helper function
 * input:
 *  the element you want to add, minadd,
 *  the predecessor, pred,
 *  the successor, succ.
 * output:
 *  none.
 * side-effects:
 *  add one element into the given position.
 */
template<class T> void DListsort<T>::add(node<T> *minadd, node<T> *pred, node<T> *succ)
{
    // add one node (element) minadd between the two elements outlastnode & dummy.
    (*minadd).setprevious(pred);
    (*minadd).setnext(succ);
    (*pred).setnext(minadd);
    (*succ).setprevious(minadd);
    return;
}
```

```
template<class T> void node<T>::setprevious(node<T> *pt)
{
    pt_previous = pt;
    return;
}
```

STATE 3


```
// update the outlastnode to be the last node of the new linked list.
// NOTE that after minnode is added, the outlastnode is NOT the OUT LAST NODE any more.
outlastnode = minnode;
```

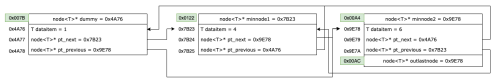
STATE 4




```
// add minnode between outlastnode & dummy.
add(minnode, outlastnode, dummy);
```

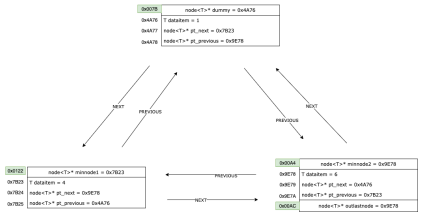
```
/* helper function
 * input:
 *  the element you want to add, minadd,
 *  the predecessor, pred,
 *  the successor, succ.
 * output:
 *  none.
 * side-effects:
 *  add one element into the given position.
 */
template<class T> void DListsort<T>::add(node<T> *minadd, node<T> *pred, node<T> *succ)
{
    // add one node (element) minadd between the two elements outlastnode & dummy.
    (*minadd).setprevious(pred);
    (*minadd).setnext(succ);
    (*pred).setnext(minadd);
    (*succ).setprevious(minadd);
    return;
}
```

STATE 5

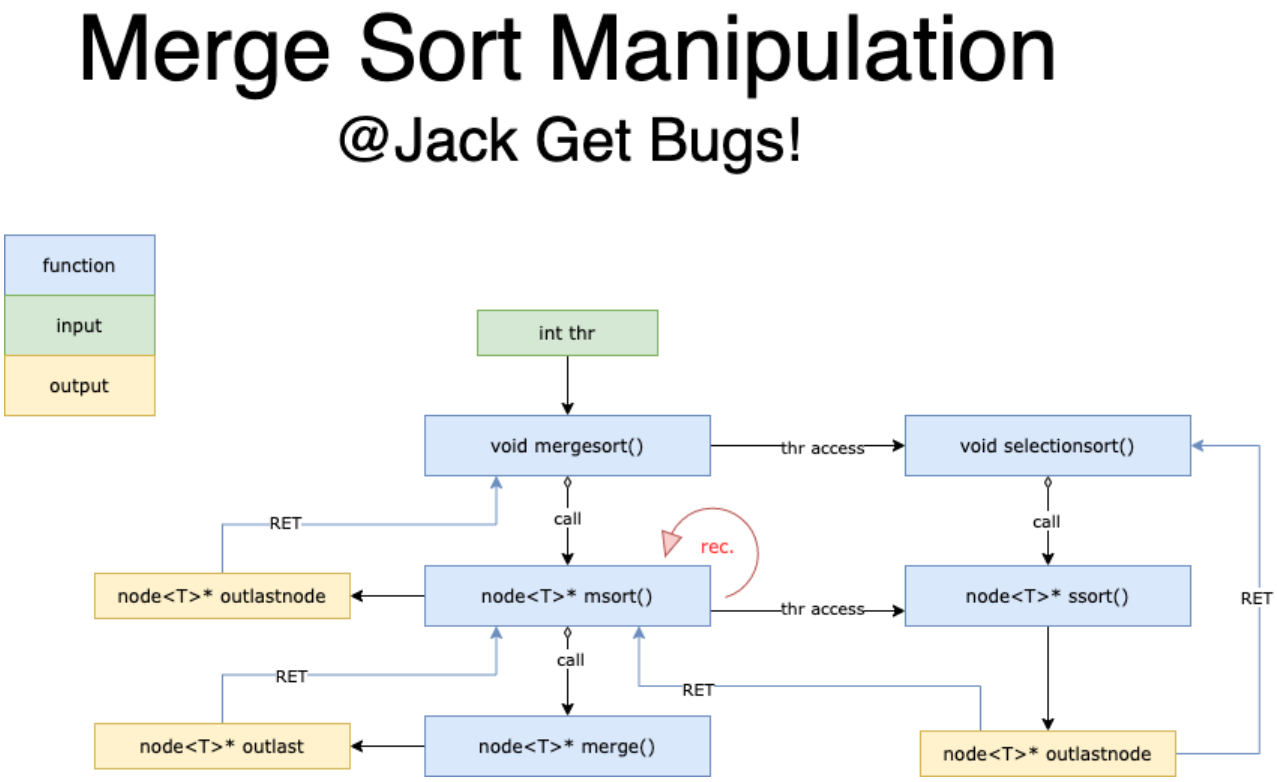



```
// update the outlastnode to be the last node of the new linked list.
// NOTE that after minnode is added, the outlastnode is NOT the OUT LAST NODE any more.
outlastnode = minnode;
```

EQUIVALENT



Merge Sort


Merge Sort Manipulation
@Jack Get Bugs!
function
input
output
int thr
void mergesort()
thr access
void selectionsort()
call
call
RET
rec.
node<T>* outlastnode
node<T>* msort()
thr access
node<T>* ssort()
RET
call
RET
RET
node<T>* outlast
node<T>* merge()
node<T>* outlastnode