What is the DQUE library?

- DQUE is a set of functions that implement a double-ended queue using a circular linked-list
- Can be used for LIFO, FIFO, and simple priority queues
- Can push and pop nodes onto/from the front or back of the queue with runtime of O(1)
- Priority queue search/insert/remove functions have a runtime of (n)

Why use the DQUE library?

- Simple and easy to use
    - Yes, yes, you've heard that before, but this really is easy (yes, you've heard that too)
    - All structures hidden from user, no need to learn how DQUE works under the covers
    - No need to integrate or adapt to user 's code, all functions used without modifications
    - Uses forward/rearward iterators instead of pointer manipulations
    - Most function names similar to C++ standard library deque/list names modified for C
    - Short function names with most functions only requiring two arguments
    - All functions return error code similar to C-11 functions with additional error checking
- Extremely flexible to use
    - DQUE routines only store a pointer to user's supplied data, so data can be anything
    - A single pointer to data can be inserted on multiple queues
    - A single queue can store multiple types of data (user responsible for tracking types)
    - All priority queue keys/comparison functions are user supplied, so keys can be anything
    - Comparison function not fixed, allows multiple key/function pairs on single queue
- Small, efficient code
    - Extensive code reuse, e.g. only one routine each to insert/remove node onto a queue
    - Efficient dynamic memory allocations that reduce possible memory leaks
    - Most functions are written in separate files so linkers only link code actually used

- DQUE library uses two data structures per queue, a single ‘qhead’ and numerous ‘qnode’s

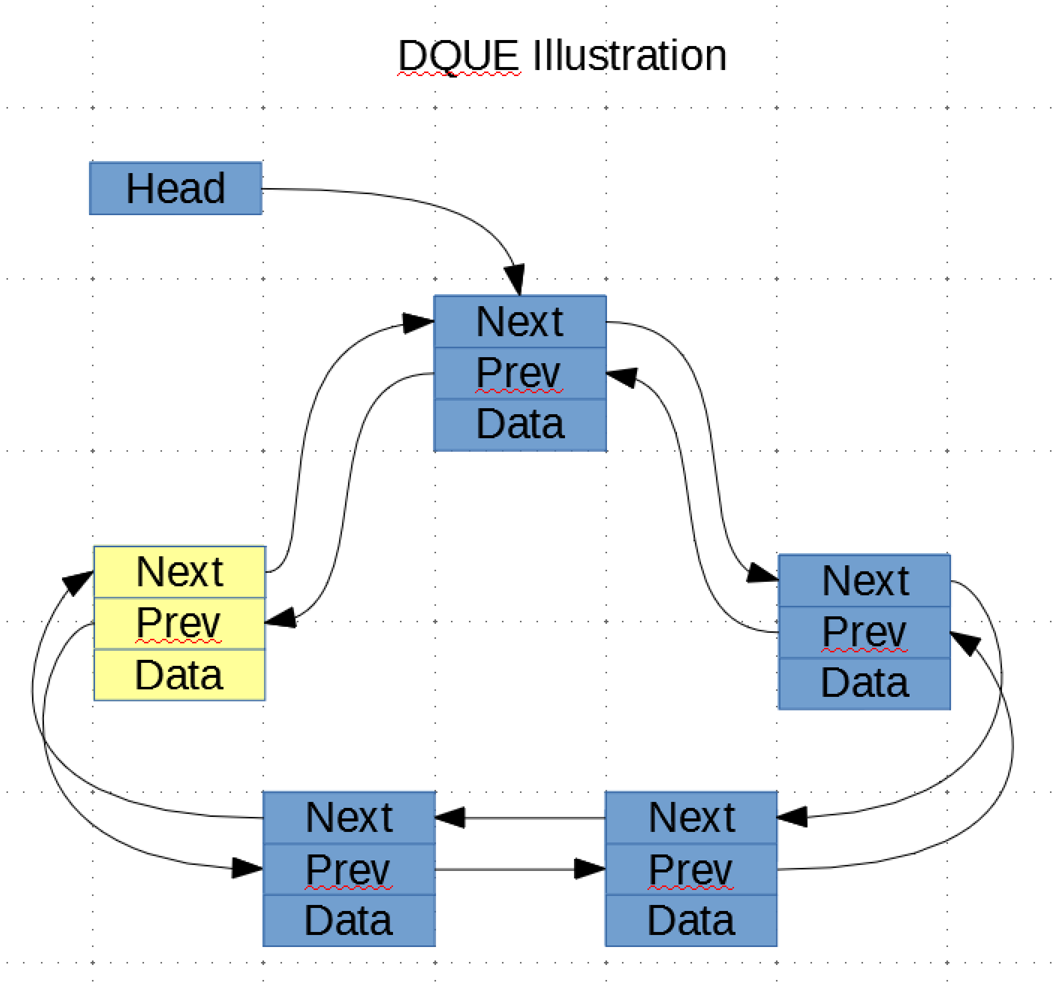
```
typedef struct qhead {
    unsigned int        vers;  /* version number for later expansion     */
    unsigned int        flgs;  /* bit flags to modify library behavior   */
    qnode              *head;  /* pointer to user’s list of nodes (queue)*/
    qnode              *free;  /* pointer to list of free nodes          */
    qnode              *blks;  /* pointer to list of allocation blocks   */
    unsigned int        hcnt;  /* number of nodes in head queue          */
    unsigned int        fcnt;  /* number of nodes in free list           */
    unsigned int        bcnt;  /* number of nodes in blks list           */
    unsigned int        acnt;  /* number of nodes to allocate            */
    } qhead, *qheadp;

typedef struct qnode {
    struct qnode       *next;  /* pointer to next node forward in queue  */
    struct qnode       *prev;  /* pointer to previous node rearward      */
    void               *data;  /* pointer to user’s data                 */
    } qnode, *qnodep;

#define qiter qnode            /* pointer to iterator == pointer to qnode*/
```

1. The qhead structure has the following members:

• `vers`	version number in the form of MMMMnnnn, where MMMM is major version number in hexadecimal, and nnnn is minor version number in hexadecimal
• `flgs`	bit flags to control library behavior (see mydque.h for details)
• `head`	pointer to first (front) node in the user’s queue (NULL if queue is empty)
• `free`	pointer to free list of available nodes for use by the user
• `blks`	pointer to list of nodes used to track qnode memory allocations
• `hcnt`	current count of nodes on head list
• `fcnt`	current count of nodes on free list
• `bcnt`	current count of nodes on block list
• `acnt`	number of nodes to allocate each time DQUE allocates memory

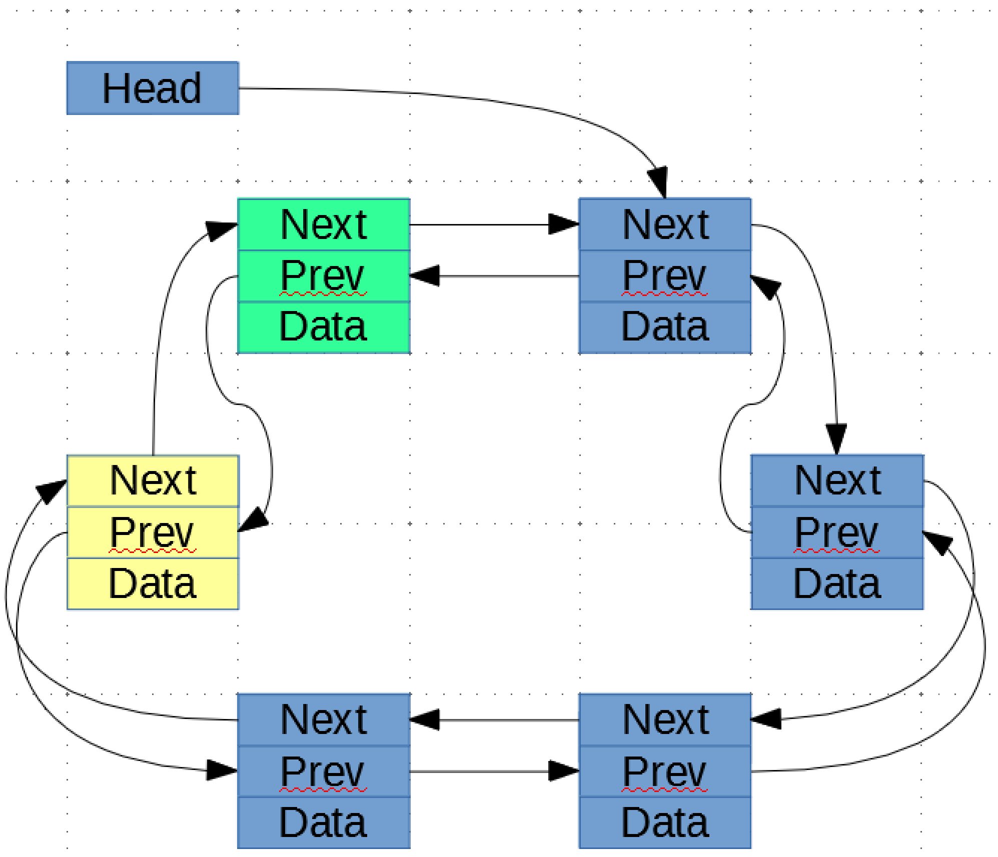
2. Note: While the qhead structure ‘head’ variable holds the official pointer to the user’s queue, it is not the only “queue”
3. A pointer to ANY qnode in the linked-list is defined as a “queue”
4. This definition is central to the understanding how the DQUE code works
5. The following charts will illustrate how this definition of queue works

1. This DQUE diagram is a circular doubly-linked list of five qnodes, each node consists of 3 pointers
   - Next is a pointer to the next qnode in the forward direction
   - Prev is a pointer to previous qnode in the rearward direction
   - Data is a pointer to the user supplied data
2. A pointer to ANY qnode in the list defines a "queue"
3. Head is such a pointer, and by definition, points to the first qnode in the "front" of the queue
4. By definition, the Head->Prev pointer points to the last qnode at the "back" of the queue (in yellow)



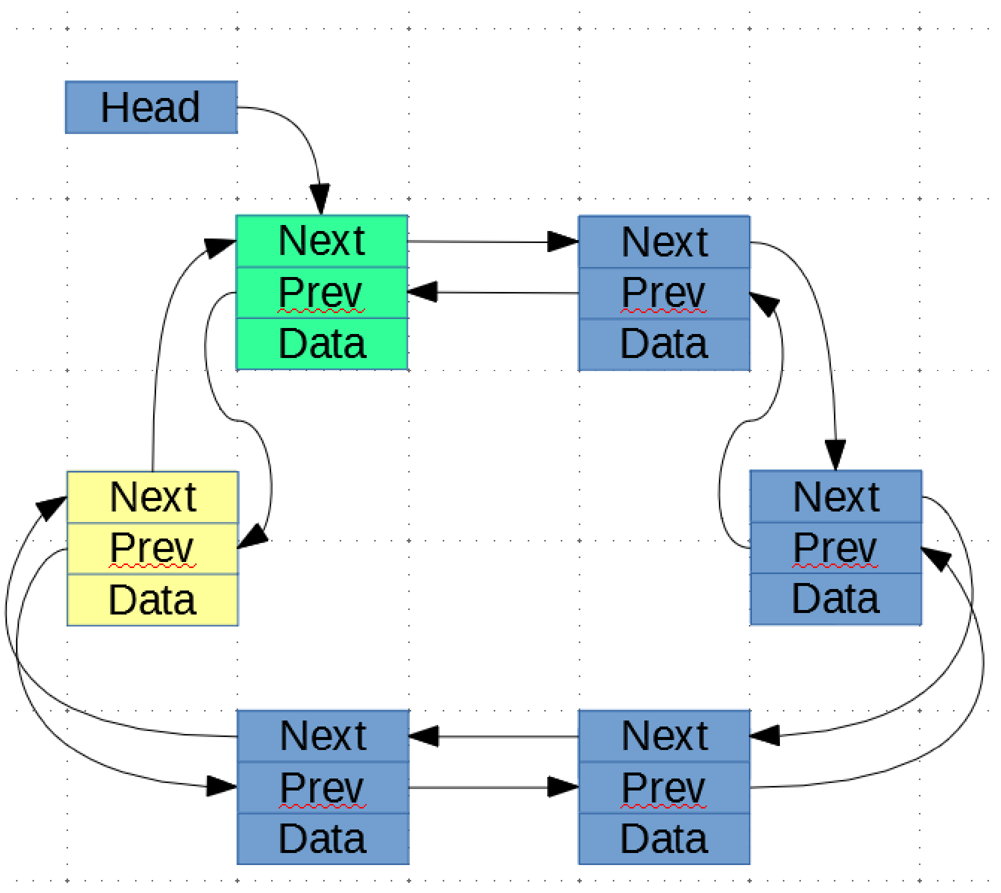
1. push_back() pulls a qnode off the free list and sets the Data pointer to specified user data
2. push_back() calls myinsert() with NOROTATE option to insert node
3. myinsert() contains the ONLY code to insert a qnode into a list
4. The myinsert() code ALWAYS inserts a new qnode at the back of the queue (colored green)
5. The Head pointer points to the same qnode, so by definition, that node is still the front of the queue
6. By definition, the green qnode becomes the new last qnode at the back of the queue
7. This diagram is the final result after calling push_back()

push_back() "pushes" node to back of queue



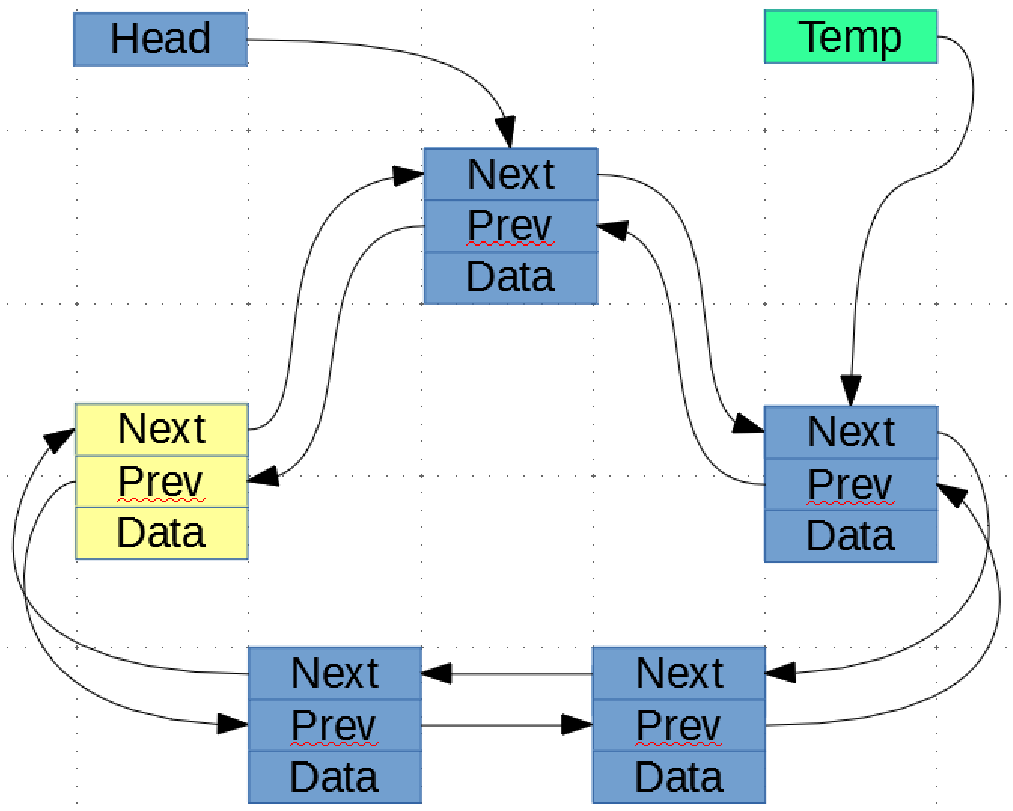
1. This diagram is the result when push_front() calls myinsert() with the ROTATE option
2. myinsert() appends the new qnode, as with push_back(), then rotates the Head pointer to point to the new qnode (colored green)
3. By definition, the green qnode becomes the new "front" of the queue while the yellow qnode remains the "back" of the queue
4. Note: myinsert() first argument is a pointer to a pointer to a qnode and **not** a pointer to the queue head structure
5. In this diagram, the first argument would be a pointer to Head
6. This allows myinsert() to be used in many situations without change

push_front() "pushes" node to front of queue

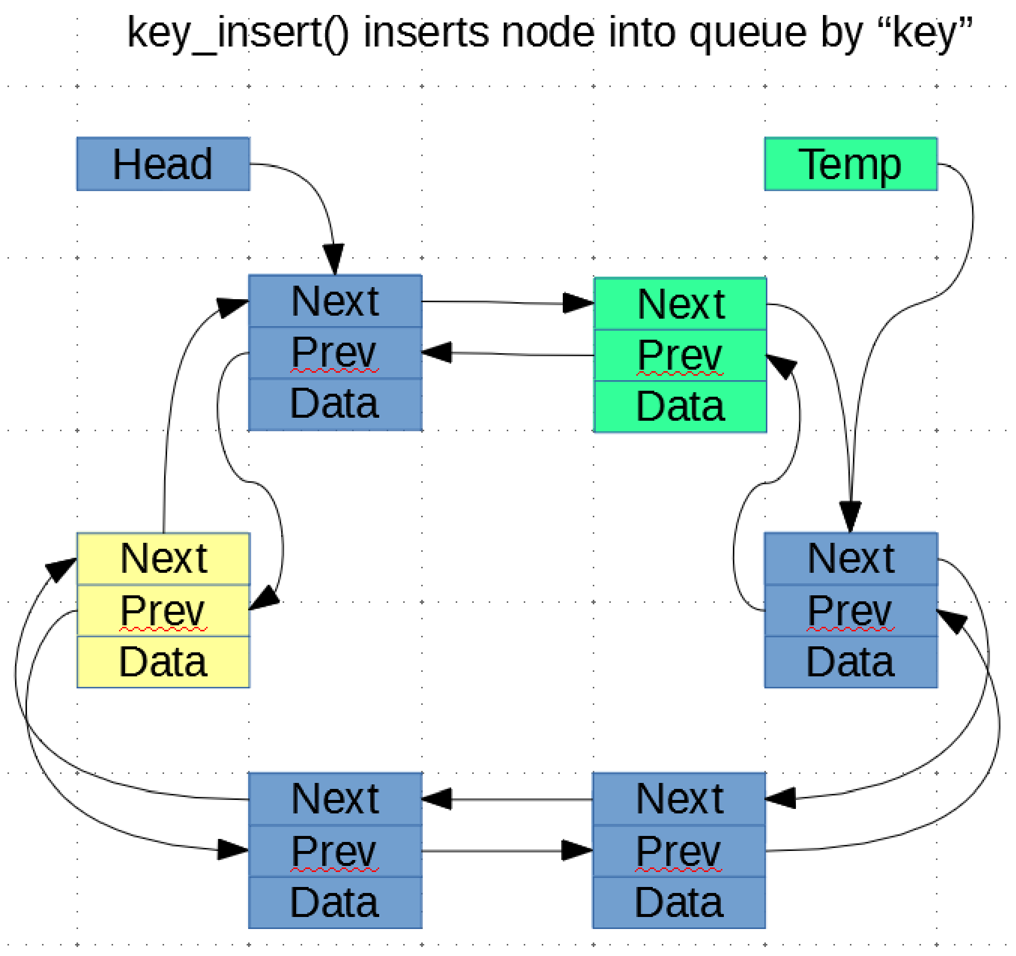


1. key_insert() calls myscan() to find where to insert the new qnode
2. myscan() returns a pointer to the first qnode in the queue after the place where the new node should be inserted (second position)
3. The returned pointer, Temp, arbitrarily points to the second node for illustrative purposes
4. This means the “key” in the new qnode is greater than or equal to the key in qnode pointed to by Head, but less than the key in the qnode pointed to by Temp
5. key_insert() then calls myinsert() with the first argument a pointer to Temp, **not** Head
6. In this case, Temp points to the “front” of a temporary “queue”

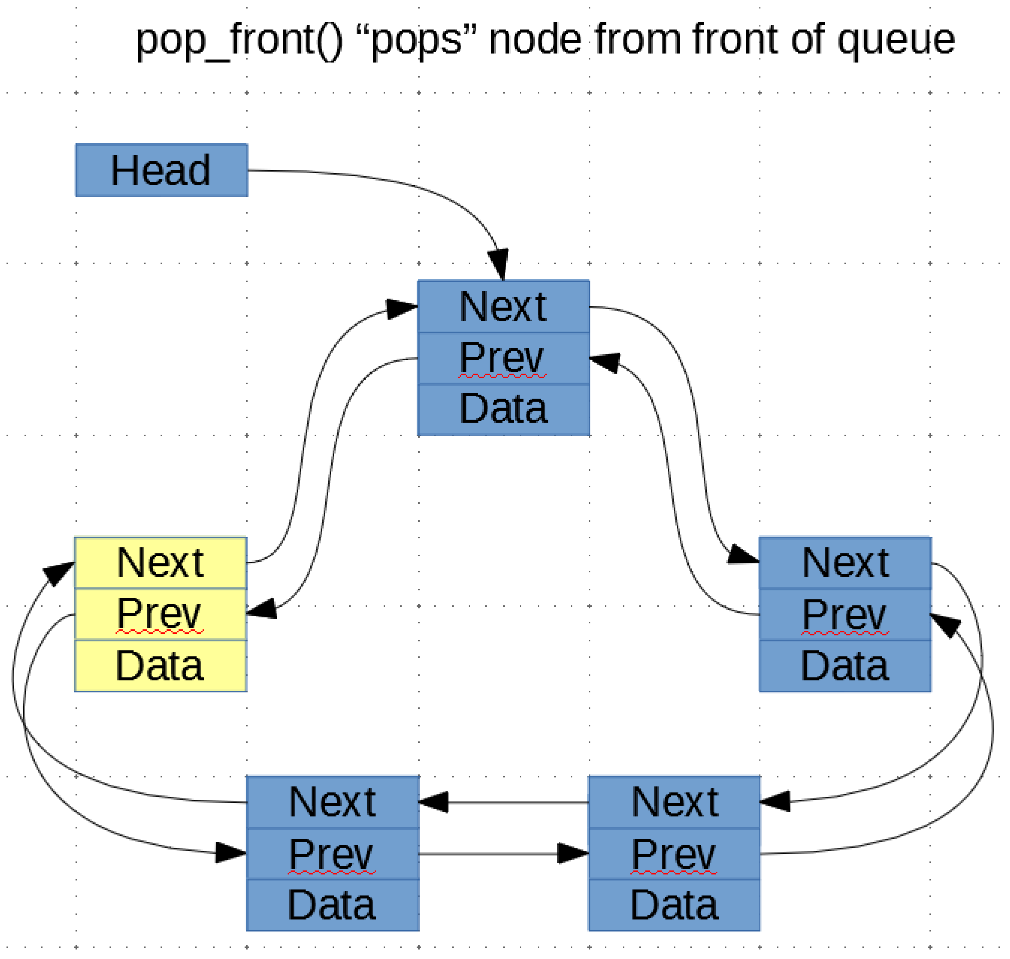
key_insert() inserts node into queue by “key”



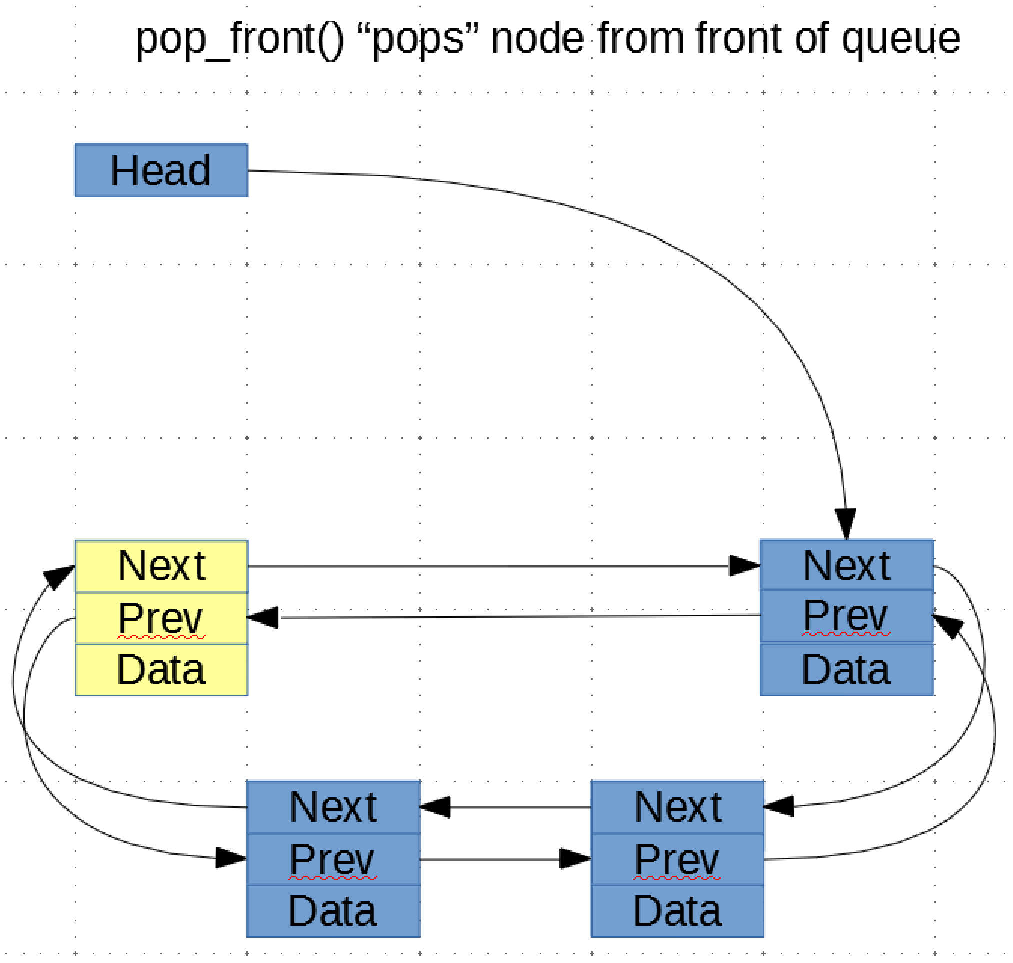
1. This diagram is the result after key_insert() calls myinsert() on Temp with the NOROTATE option
2. The Temp pointer still points to the “front” of the temporary “queue”, and the new qnode (colored green) has been inserted at the “back” of the “queue”
3. The Head pointer still points to the front of the real queue and the new qnode is inserted in the second position where myscan() indicated it should be inserted
4. Note: All comparisons must use the user supplied comparison function to compare the user data “keys” in all nodes



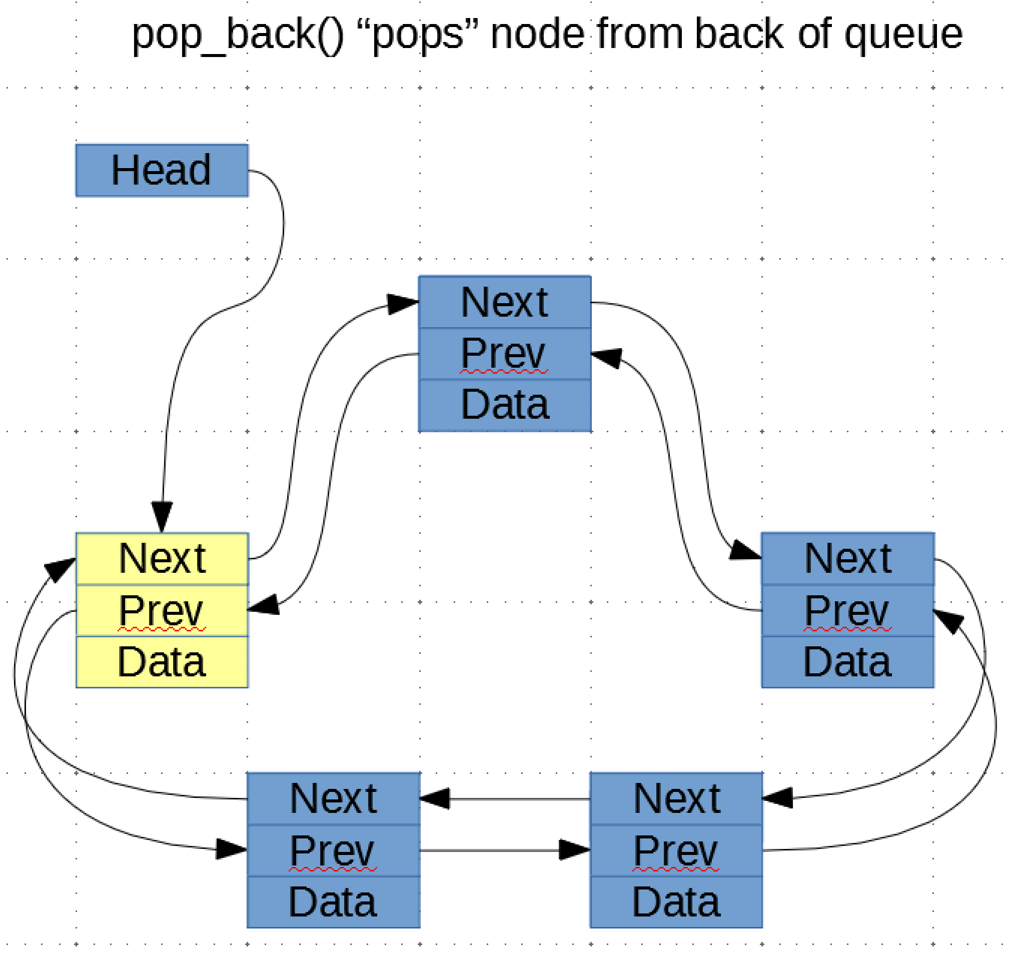
1. pop_front() calls mydelete() with NOROTATE option to remove the qnode from the front of the queue
2. mydelete() contains the ONLY code to remove a node from a list
3. The mydelete() code ALWAYS removes the qnode from the front of a queue
4. In this case, Head points to the front of the queue and the qnode to be removed
5. The last node at the back of the queue is colored yellow
6. pop_front() will call mydelete() with the first argument a pointer to Head



1. This diagram is the result after pop_front() calls mydelete() with the NOROTATE option.
2. The Head pointer now points to the new “front” of the queue (old second node)
3. The last node (back) of the queue is the same as before the removal
4. The user Data pointer from the removed qnode is returned to the user
5. The removed qnode is inserted onto the free list of the queue head structure
6. The Data pointer of the removed qnode is set to NULL to indicate that the qnode is no longer valid

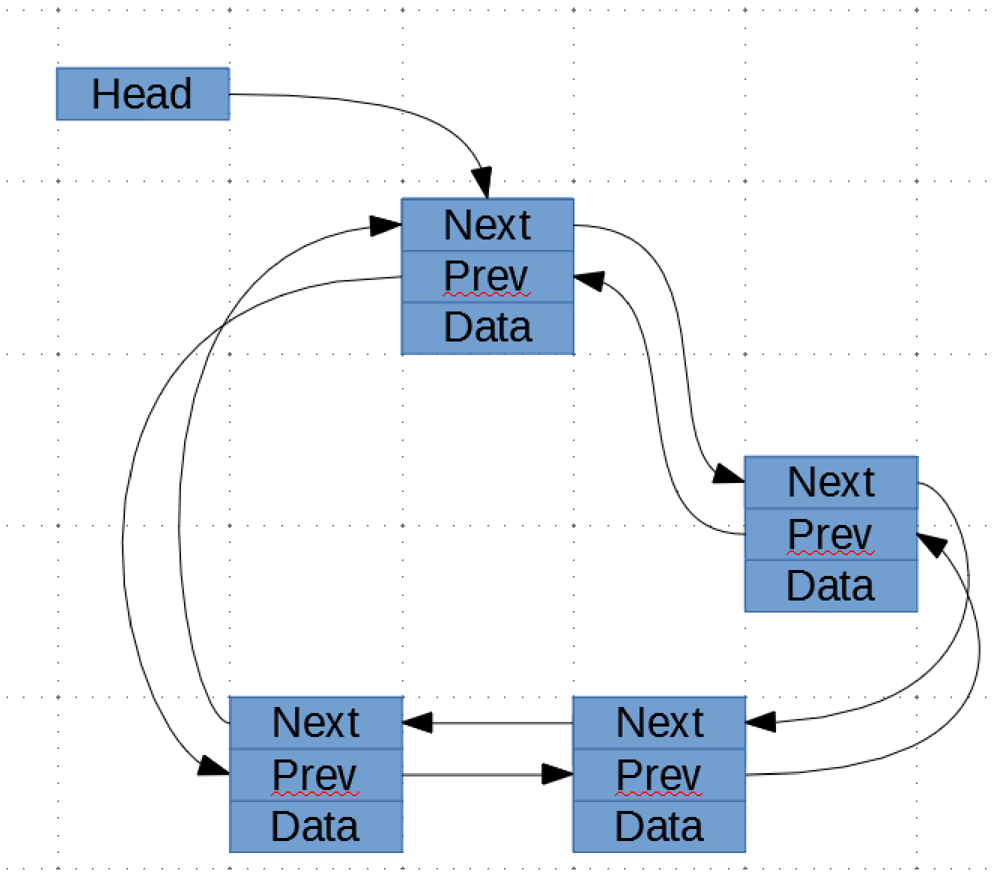


1. pop_back() will call mydelete() with the first argument a pointer to Head with the ROTATE option
2. mydelete() rotates the Head pointer to point to the last qnode (colored yellow) in the queue
3. This diagram shows the result after the rotation but before the qnode removal
4. mydelete() then removes the “first” node of the temporary “queue” as before in pop_front()



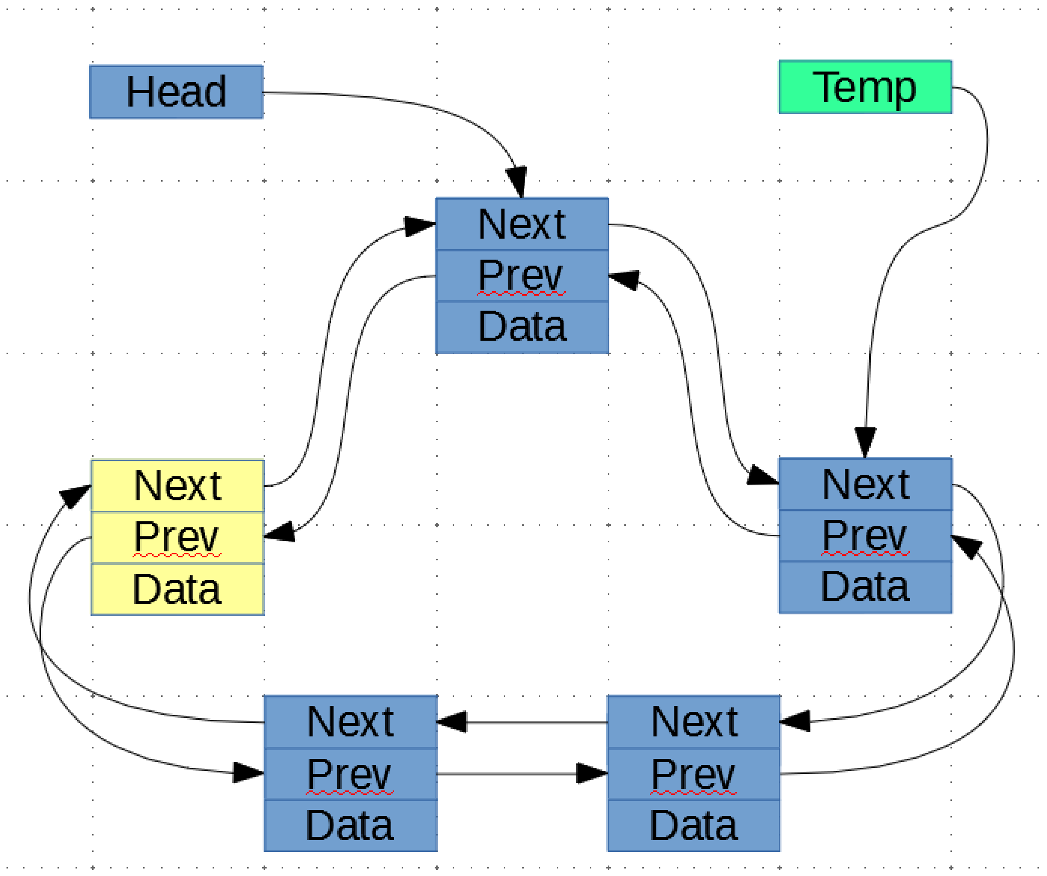
1. This diagram shows the result after removing the “first” node
2. The Head pointer now points to the “second” qnode which is the original first node (front) of queue
3. The new last qnode is the old second to last qnode
4. The user Data pointer from the removed qnode is returned to the user
5. The removed qnode is inserted onto the free list of the queue head structure
6. The Data pointer of the removed qnode is set to NULL to indicate that the qnode is no longer valid

pop_back() “pops” node from back of queue



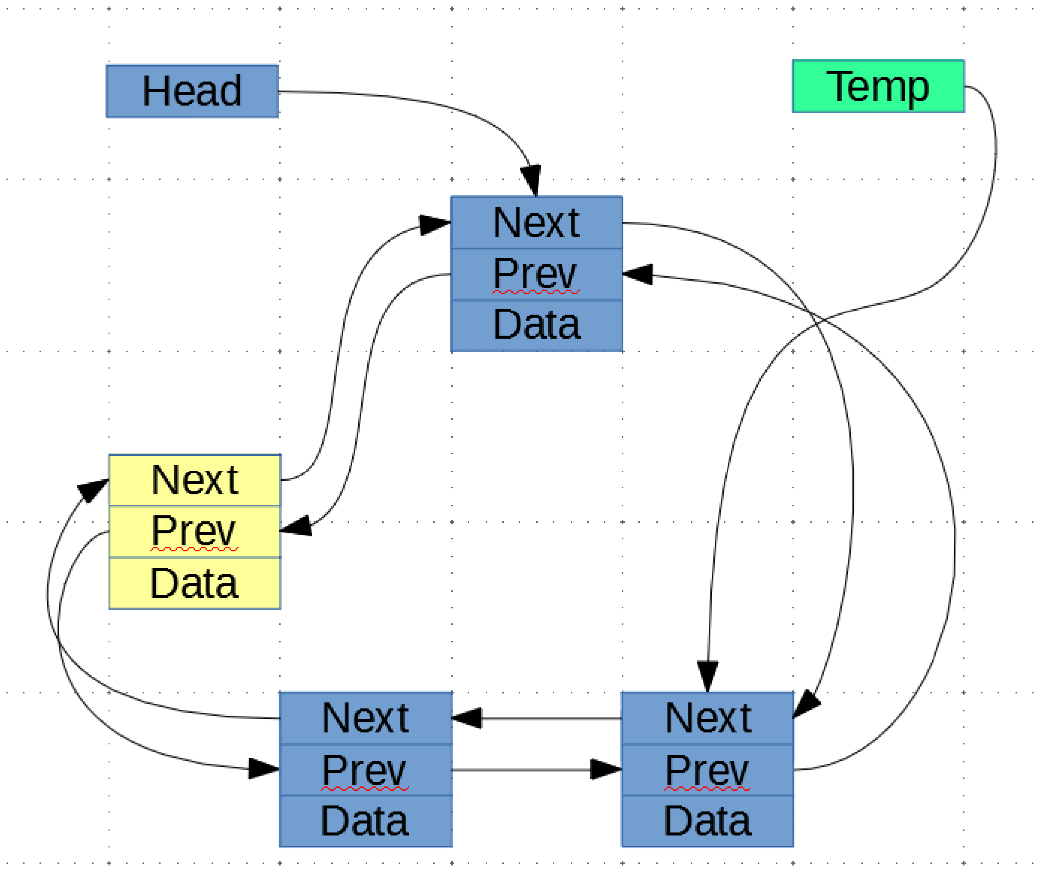
1. key_erase() calls myscan() to find the qnode in the queue with the specified "key"
2. myscan() returns a pointer to the qnode in the queue with the specified key using the user specified comparison function
3. The returned pointer, Temp, arbitrarily points to the second node in the real queue for illustrative purposes
4. key_erase() then calls mydelete() with the first argument a pointer to Temp, **not** Head
5. In this case, Temp points to the "front" of the temporary "queue"

key_erase() removes node from queue by "key"

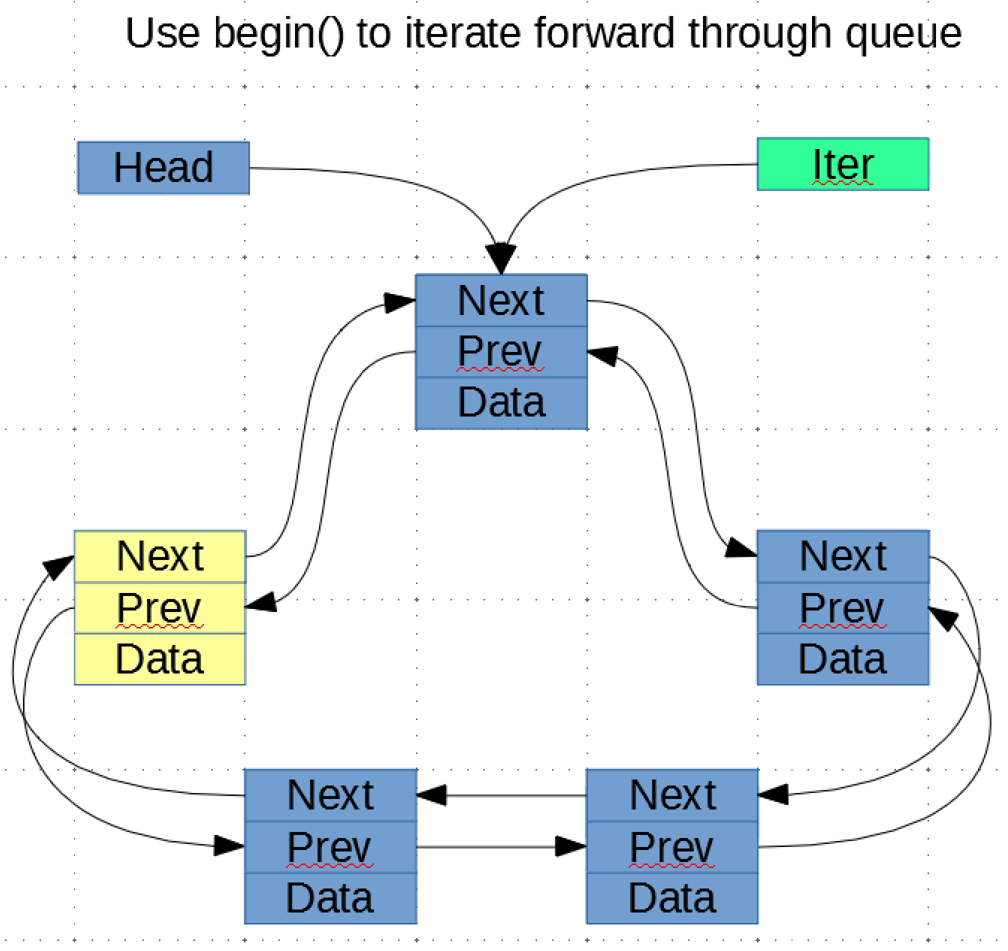


1. Diagram shows the result after key_erase() calls mydelete() with the NOROTATE option
2. The old second qnode in the queue has been removed
3. The Head pointer still points to the first (front) node in the queue and the yellow node is still the last (back) node of the queue
4. The user Data pointer from the removed qnode is returned to the user
5. The removed qnode is inserted onto the free list of the queue head structure
6. The Data pointer of the removed qnode is set to NULL to indicate that this qnode is no longer valid

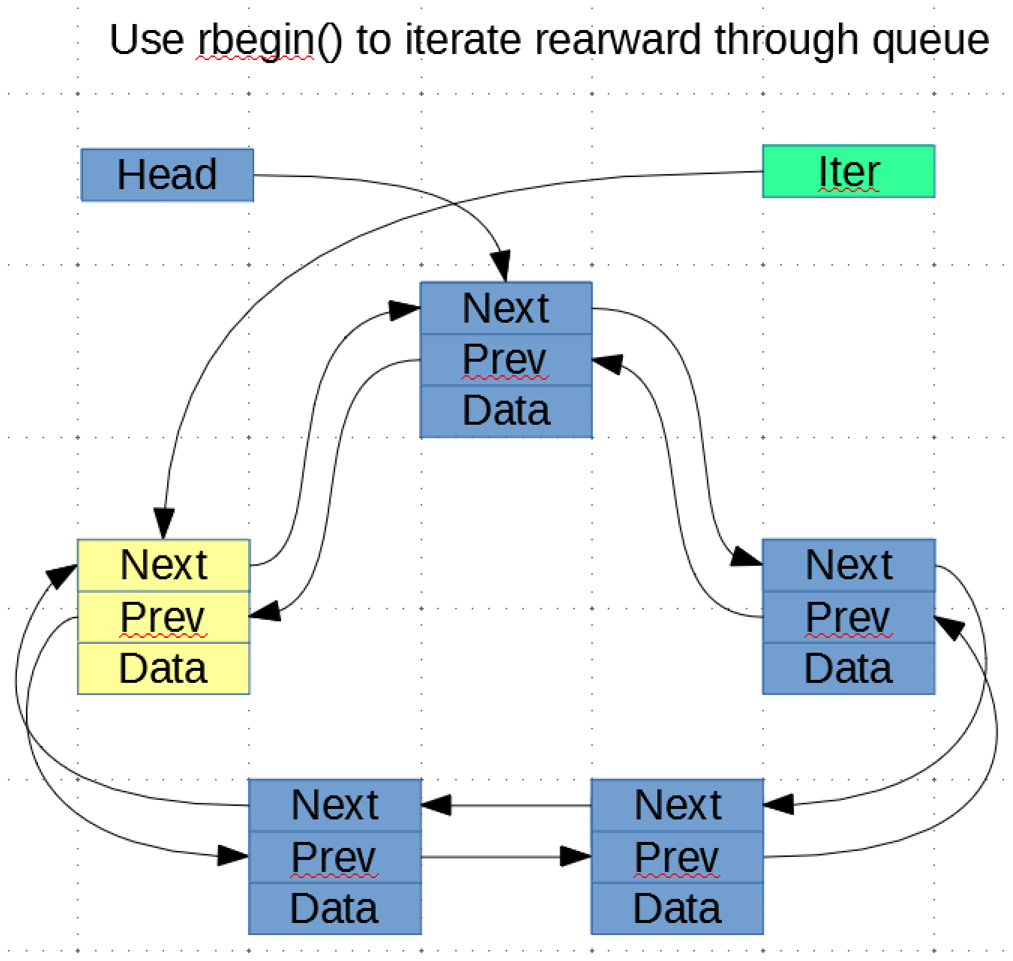
key_erase() removes node from queue by "key"



1. An iterator is nothing more than pointer to a qnode (yes, a queue)
2. begin() returns an “iterator” that points to the first (front) qnode of the queue (colored green)
3. To iterate through the queue, the user must use the next() function to get the next iterator forward from the current iterator
4. Since there is no actual “end” to a circular list, the end() function always returns a NULL pointer
5. The next() function returns NULL if the current iterator points to the last (back) node of the queue
6. The get_data() function returns the user Data pointer from the iterator

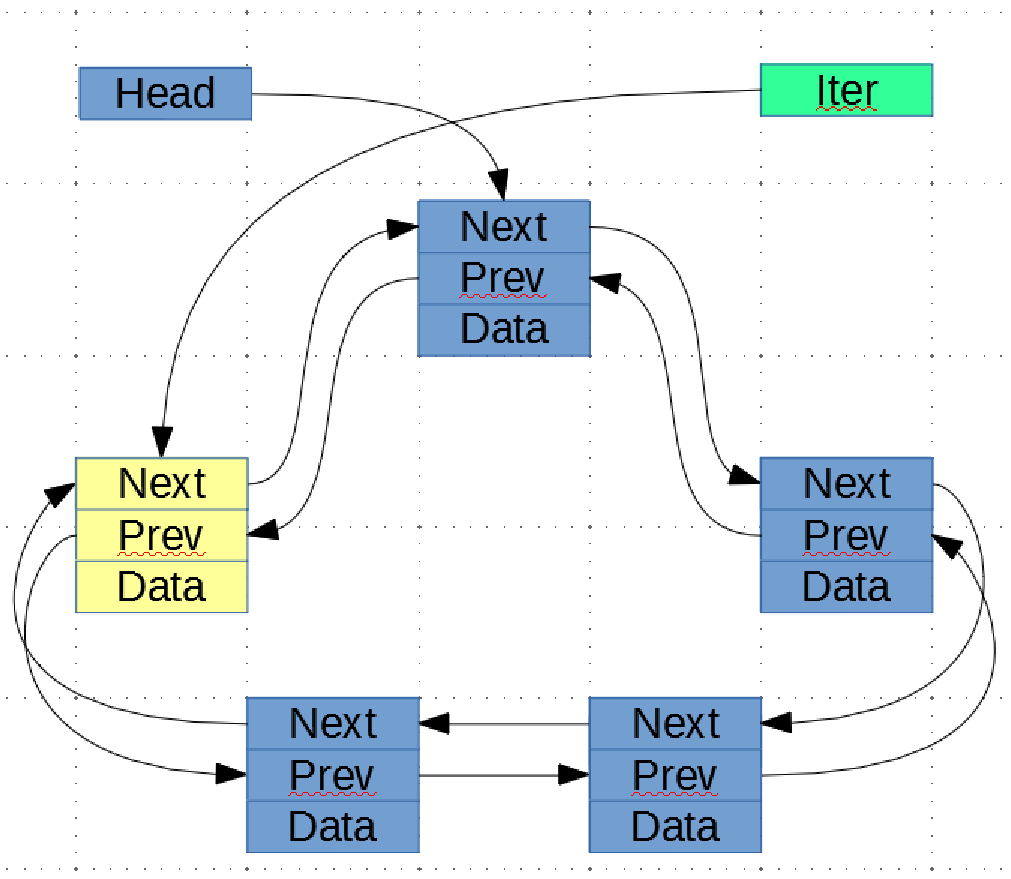


1. rbegin() returns an “iterator” that points to the last (back) qnode of the queue (colored yellow)
2. To iterate through the queue, the user must use the rnext() function to get the next iterator rearward from the current iterator
3. Since there is no actual “end” to a circular list, so the rend() function returns a NULL pointer
4. The rnext() function returns NULL if the current (iterator) points to the first (front) node of the queue
5. The get_data() function returns the user Data pointer from the iterator



1. Since an iterator is just a pointer to a qnode, the user may use both next() and rnext() with the same iterator
2. Since both end() and rend() return a NULL pointer, any iterator will terminate correctly with the next() or rnext() call appropriately
3. After the insertion/removal of any qnodes, an iterator may, or may not, be valid if the qnodes are “behind” the current position of the iterator
4. If an iterator points to a qnode that has been removed from the queue, next() and rnext() will return an error and a NULL pointer

Use rbegin() to iterate rearward through queue



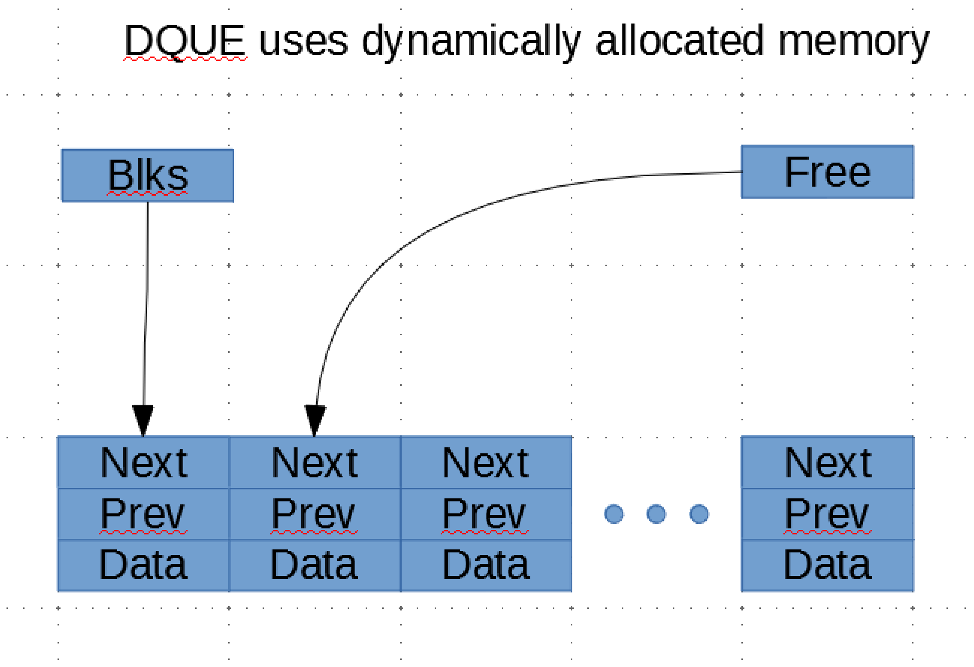
- Iterator’s are robust but cumbersome to use and can be simplified with the use of a global error code variable and some wrapper code
- Wrapper functions return a null pointer on error, check global error variable for error

```
int errcode;

qiter *
mybegin( qhead *queue ) {
    qiter *iter = NULL_QITER;

    if ((errcode = begin( queue, (qiter **)&iter )) != DQUEERR_NOERR) {
        if ((errcode = end( queue, (qiter **)&iter )) != DQUEERR_NOERR) {
            iter = NULL_QITER;
        }
    }
    return (iter);
}

qiter *
myend( qhead *queue ) {
    qiter *iter = NULL_QITER;

    if ((errcode = end( queue, (qiter **)&iter )) != DQUEERR_NOERR) {
        iter = NULL_QITER;
    }
    return (iter);
}

qiter *
mynext( qhead *queue, qiter **iterp ) {
    if ((errcode = next( queue, iterp )) != DQUEERR_NOERR) {
        if ((errcode = end( queue, iterp )) != DQUEERR_NOERR) {
            *iterp = (qiter *)NULL;
        }
    }
    return (*iterp);
}
```

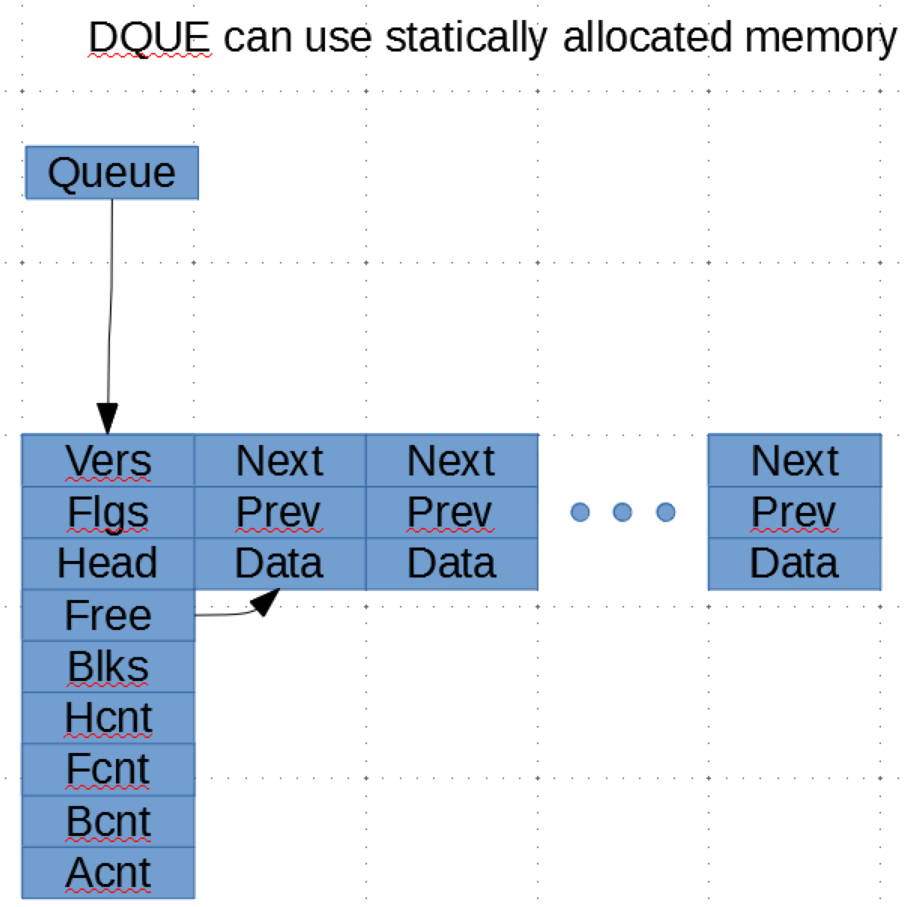
- The mydata() function could be customized for specific user data structures

```
void *
mydata( qiter *iter ) {
    void *data = (void *)NULL;

    if ((errcode = get_data( iter, (void **)&data )) != DQUEERR_NOERR) {
        data    = (void *)NULL;
    }
    return (data);
}

// example of using wrapper functions
qiter *iter;
for (iter = mybegin(queue); iter != myend(queue); mynext(queue, &iter)) {
    if ((ptr = mydata( iter )) == (void *)NULL) {
         // handle error using global error code
    } else {
         // do something with data
    }
}
```

1. The create() function normally allocates memory for the qhead structure
2. If an insert function finds the Free list empty, it will call myalloc() to allocate more memory for qnodes
3. myalloc() allocates a block of memory to create qnodes
4. The size of the block is equal to 25 * sizeof(qnode) by default, but can be changed using options()
5. The first qnode has the same memory address as the whole block, so the first qnode is saved on the Blks list to be used later by destroy() to free the blocks
6. The remaining 24 qnodes are put on the Free list for the user



1. The create() function can be given a block of memory for all allocations, including the qhead. This sets a bit flag to prevent all future dynamic allocations
2. Now if an insert function finds the Free list empty, it will call myalloc() which will always FAIL
3. It is the user's responsibility to insure that there are qnodes available on the Free list
4. The user can use the options() function to give DQUE more memory for qnodes
5. The Blks list is not used since there are no dynamic allocations
6. The destroy() function does not free any memory; it is the user's responsibility to free the memory



1. The options() function is used to control the behavior of each queue:

- **`VERSION`** returns major and minor version numbers in decimal
- **`ALLOCSZ`** sets number of qnodes to allocate each dynamic memory block allocation
- **`HEADSIZ`** returns the size of the qhead structure
- **`NODESIZ`** returns the size of the qnode structure
- **`NOALLOC`** with no dynamic allocation, allows user to add memory blocks for qnodes
- **`NODUPE`** non-zero argument means no duplicate qnodes allowed in priority queue
- **`NOSCAN`** non-zero argument means no scan of queue with insert() and remove()

2. The options() function only affects the queue used as the first argument to options()
3. The options HEADSIZ, and NODESIZ can be sent a NULL qhead pointer. These values can be used to calculate static memory allocations when not using dynamic memory
4. The NOALLOC option does **not** turn off dynamic allocation. Supplying a memory block when calling create() turns off dynamic allocation. This option is used to supply additional memory blocks to use for qnodes
5. The NOSCAN option changes insert() and remove() execution from O(n) to O(1), but does not detect inserting or removing qnodes from the wrong queue

1. All DQUE functions return a status similar to functions introduced with C11 standard
2. A returned error status of zero means no errors were detected
3. Error codes are small positive integers, deliberately not listed, and may change in future
4. Error codes may be translated into a character string using error()
5. error() is very similar in function to standard C library strerror_s()
6. error(err, buf, siz) takes three arguments
    a. err - Error code that is returned by another DQUE function
    b. buf - Buffer for returned translated error message
    c. siz - Size of buf, limits size of returned message to siz - 1
7. As a convenience, err may be negative and error() will automatically negate it before use
8. To insure adequate buffer length, error_len() returns length of a specific error message

1. A normal priority queue comparison function creates a queue in ascending order, but I need a queue in descending order?

```
/* integer comparison function, others similar */
int intcompdec( void *i1, void *i2 ) {
     /* reverse arguments for descending order queue */
     return (*(int *)i2 - *(int *)i1);
}
```

2. I need to search a priority queue but the queue is not ordered by the field I need to search and is therefore unordered. How do I search the whole queue?

```
typedef struct {
     int  id;
     char name[20];
     int  year;
     int  classes[7];
} record;

typedef struct {
     int   idx;
     int   val;
     char *str;
} reckey;
```

```
/* This shows a more complicated "key" and that the key */
/* can be almost anything.                               */
/* NOTE: that reccomp returns zero if the key and qnode */
/* data match, and a value greater than zero to insure */
/* that reccomp searches the entire queue. */
/* This is required when searching a queue on a field */
/* other than the field on which it is ordered. */
int reccomp( void *keyval, void *recval ) {
    reckey *key = (reckey *)keyval;
    record *rec = (record *)recval;
    int  retval = 1; /* return +val to search whole queue*/

    switch (key->idx) { /* set idx to select search key */
        case 0:
          if (key->val == rec->id) { retval = 0; }
          break;
        case 1:
          if (!strcmp(key->str, rec->name)) { retval = 0; }
          break;
        case 2:
          if (key->val == rec->year){ retval = 0; }
          break;
    }

    return (retval);
}
```

3. I have an unordered FIFO queue with several types of data. How do I search only one type of data?

```
/* This is basically the same as question #2, except */
/* both the “key” and comparison function need to pass */
/* which type of data to search and only compare the */
/* “key” and node data if the type matches */
```