cs280s21-b.sg

Grade

Reviewed on Monday, February 15, 2021, 9:10 AM by Automatic grade
grade: 100.00 / 100.00

Assessment report[-]
[+]**Summary of tests**

Submitted on Monday, February 15, 2021, 9:10 AM (Download)
BList.h

```
1  ////////////////////////////////////////////////////////////////////////////
2  #ifndef BLIST_H
3  #define BLIST_H
4  ////////////////////////////////////////////////////////////////////////////
5
6  #include <string> // error strings
7
8  /*!
9    The exception class for BList
10 */
11 class BListException : public std::exception
12 {
13   private:
14     int m_ErrCode;             //!< One of E_NO_MEMORY, E_BAD_INDEX,E_DATA_ERROR
15     std::string m_Description; //!< Description of the exception
16
17   public:
18     /*!
19       Constructor
20
21       \param ErrCode
22         The error code for the exception.
23
24       \param Description
25         The description of the exception.
26     */
27     BListException(int ErrCode, const std::string& Description) :
28     m_ErrCode(ErrCode), m_Description(Description) {};
29
30     /*!
31       Get the kind of exception
32
33       \return
34         One of E_NO_MEMORY, E_BAD_INDEX, E_DATA_ERROR
35     */
36     virtual int code() const {
37       return m_ErrCode;
38     }
39
40     /*!
41       Get the human-readable text for the exception
42
43       \return
44         The description of the exception
45     */
46     virtual const char *what() const throw() {
47       return m_Description.c_str();
48     }
49
50     /*!
51       Destructor is "implemented" because it needs to be virtual
52     */
53     virtual ~BListException() {
54     }
55
56     //! The reason for the exception
57     enum BLIST_EXCEPTION {E_NO_MEMORY, E_BAD_INDEX, E_DATA_ERROR};
58 };
59
60 /*!
61   Statistics about the BList
62 */
63 struct BListStats
64 {
65     //!< Default constructor
66   BListStats() : NodeSize(0), NodeCount(0), ArraySize(0), ItemCount(0)  {};
67
68   /*!
69     Non-default constructor
70
71     \param nsize
72       Size of the node
73
74     \param ncount
75       Number of nodes in the list
76
77     \param asize
78       Number of elements in each node (array)
79
80     \param count
81       Number of items in the list
82
83   */
84   BListStats(size_t nsize, int ncount, int asize, int count) :
85   NodeSize(nsize), NodeCount(ncount), ArraySize(asize), ItemCount(count)  {};
86
87   size_t NodeSize; //!< Size of a node (via sizeof)
88   int NodeCount;   //!< Number of nodes in the list
89   size_t ArraySize;   //!< Max number of items in each node
90   int ItemCount;   //!< Number of items in the entire list
91 };
92
93 /*!
94   The BList class
95 */
96 template <typename T, unsigned Size = 1>
97 class BList
98 {
99
100   public:
101     /*!
102       Node struct for the BList
103     */
104     struct BNode
105     {
106       BNode *next;    //!< pointer to next BNode
107       BNode *prev;    //!< pointer to previous BNode
```

```
    unsigned count;      //!< number of items currently in the node
    T values[Size]; //!< array of items in the node

    //!< Default constructor
    BNode() : next(0), prev(0), count(0) {}
  };

  BList();                              // default constructor
  BList(const BList &rhs);              // copy constructor
  ~BList();                             // destructor
  BList& operator=(const BList &rhs); // assign operator

    // arrays will be unsorted, if calling either of these
  void push_back(const T& value);
  void push_front(const T& value);

    // arrays will be sorted, if calling this
  void insert(const T& value);

  void remove(int index);
  void remove_by_value(const T& value);

  int find(const T& value) const;       // returns index, -1 if not found

  T& operator[](int index);             // for l-values
  const T& operator[](int index) const; // for r-values

  size_t size() const;   // total number of items (not nodes)
  void clear();          // delete all nodes

  static size_t nodesize(); // so the allocator knows the size

    // For debugging
  const BNode *GetHead() const;
  BListStats GetStats() const;

 private:
    BNode *head_; //!< points to the first node
    BNode *tail_; //!< points to the last node

    BListStats BListStats_;

    bool sort;
    // Other private data and methods you may need ...

    BNode* CreateNode();
    BNode* SplitNode(BNode* node, const T value);

    void InsertFront(BNode* node, const T value);
    void InsertBack(BNode* node, const T value);
    void InsertMiddle(BNode* node, const T value);
};

#include "BList.cpp"

#endif // BLIST_H
```

BList.cpp

```
/*****************************************************************************/
/*!
\file:      BList.cpp
\author:    Goh Wei Zhe, weizhe.goh, 440000119
\par email: weizhe.goh\@digipen.edu
\date:      February 11, 2021
\brief      To implement an API for Abstract Data Type (ADT)

Copyright (C) 2021 DigiPen Institute of Technology.
Reproduction or disclosure of this file or its contents without the
prior written consent of DigiPen Institute of Technology is prohibited.
*/
/*****************************************************************************/
#include "BList.h"

/*****************************************************************************/
/*!
\fn     template <typename T, unsigned Size>
        size_t BList<T, Size>::nodesize(void)

\brief  Templated function to return size of a node

\return Returns size of a node (via sizeof)
*/
/*****************************************************************************/
template <typename T, unsigned Size>
size_t BList<T, Size>::nodesize(void)
{
  return sizeof(BNode);
}

/*****************************************************************************/
/*!
\fn     template <typename T, unsigned Size>
        const typename BList<T, Size>::BNode* BList<T, Size>::GetHead() const

\brief  Templated function to return the head pointer

\return Returns the head pointer
*/
/*****************************************************************************/
template <typename T, unsigned Size>
const typename BList<T, Size>::BNode* BList<T, Size>::GetHead() const
{
  return head_;
}

/*****************************************************************************/
/*!
\fn     template <typename T, unsigned Size>
        BList<T, Size>::BList():head_{nullptr}, tail_{nullptr}, sort{true}

\brief  Constructor to initialise class variables

*/
/*****************************************************************************/
template <typename T, unsigned Size>
BList<T, Size>::BList():head_{nullptr}, tail_{nullptr}, sort{true}
{
    BListStats_.NodeSize = nodesize();
    BListStats_.ArraySize = Size;
    BListStats_.NodeCount = 0;
    BListStats_.ItemCount = 0;
}

/*****************************************************************************/
/*!
\fn     template <typename T, unsigned Size>
        BList<T, Size>::BList(const BList& rhs)

\brief  Copy Constructor

\param  rhs - The object to be copied from
*/
/*****************************************************************************/
template <typename T, unsigned Size>
BList<T, Size>::BList(const BList& rhs)
{
    head_ = NULL;
    tail_ = NULL;

    sort = rhs.sort;
    BListStats_.NodeSize = nodesize();
    BListStats_.ArraySize = Size;
    BListStats_.NodeCount = 0;
    BListStats_.ItemCount = 0;

    *this = rhs;
}

/*****************************************************************************/
/*!
\fn     template<typename T, unsigned Size>
        BList<T, Size>::~BList()

\brief  Destructor

*/
/*****************************************************************************/
template<typename T, unsigned Size>
BList<T, Size>::~BList()
{
    clear();
}

/*****************************************************************************/
/*!
```

```
\fn     template<typename T, unsigned Size>
        BList<T, Size>& BList<T, Size>::operator=(const BList<T, Size>& rhs)

\brief  Assignment Operator

\param  rhs - Object to be copied from

\return BList reference

*/
/******************************************************************************/
template<typename T, unsigned Size>
BList<T, Size>& BList<T, Size>::operator=(const BList<T, Size>& rhs)
{
    clear();

    //Set new stats variable
    BListStats_.ItemCount = rhs.BListStats_.ItemCount;
    BListStats_.NodeCount = rhs.BListStats_.NodeCount;
    BListStats_.ArraySize = rhs.BListStats_.ArraySize;

    BNode* NodePtr_rhs = rhs.head_;
    BNode* NodePtr = head_;


    BNode* node = CreateNode();
    head_ = node;
    NodePtr = head_;

    //Copy data from rhs head into new head
    for(unsigned i = 0; i < NodePtr_rhs->count; ++i)
    {
        NodePtr->values[i] = NodePtr_rhs->values[i];
        ++NodePtr->count;
    }

    NodePtr_rhs = NodePtr_rhs->next;

    //Loop through the remaining list
    while(NodePtr_rhs)
    {
        BNode* newNode = CreateNode();

        //If last node in list, reset pointers
        if(NodePtr_rhs == rhs.tail_)
        {
            tail_ = newNode;
            newNode->prev = NodePtr;

            NodePtr->next = newNode;
            NodePtr = NodePtr->next;
        }
        //If middle of list, reset pointers
        else
        {
            newNode->prev = NodePtr;
            NodePtr->next = newNode;
            NodePtr = NodePtr->next;
        }

        //Copy data from rhs to this node
        for(unsigned i = 0; i < NodePtr_rhs->count; ++i)
        {
            NodePtr->values[i] = NodePtr_rhs->values[i];
            ++NodePtr->count;
        }

        NodePtr_rhs = NodePtr_rhs->next;
    }

    return *this;
}

/******************************************************************************/
/*!
\fn     template<typename T, unsigned Size>
        void BList<T, Size>::push_back(const T& value)

\brief  Templated function to push back nodes to the back of the list

\param  value - value of node to be inserted

*/
/******************************************************************************/
template<typename T, unsigned Size>
void BList<T, Size>::push_back(const T& value)
{
    //Unsorted
    sort = false;

    //If head is empty
    if(head_ == nullptr)
    {
        head_ = CreateNode();
        tail_ = head_;

        head_->next = NULL;
        head_->next = NULL;

        head_->values[0] = value;
        ++head_->count;

        ++BListStats_.NodeCount;
        ++BListStats_.ItemCount;

        return;
    }
```

```
215
216     //Make nodeHead point be tail
217     BNode* nodeHead = tail_;
218
219     if(nodeHead->count == BListStats_.ArraySize)
220     {
221         BNode* newNode = CreateNode();
222
223         newNode->values[0] = value;
224         ++newNode->count;
225
226         //Reset pointers, make this node the tail
227         tail_->next = newNode;
228         newNode->prev = tail_;
229         newNode->next = NULL;
230         tail_ = newNode;
231
232         ++BListStats_.NodeCount;
233     }
234     else
235     {
236         nodeHead->values[nodeHead->count] = value;
237         ++nodeHead->count;
238     }
239
240     ++BListStats_.ItemCount;
241 }
242
243 /******************************************************************************/
244 /*!
245 \fn     template<typename T, unsigned Size>
246         void BList<T, Size>::push_front(const T& value)
247
248 \brief  Templated function to insert nodes from the front of the list
249
250 \param  value - value of node to be inserted
251
252 */
253 /******************************************************************************/
254 template<typename T, unsigned Size>
255 void BList<T, Size>::push_front(const T& value)
256 {
257     //unsorted
258     sort = false;
259
260     //if node is empty
261     if(head_ == nullptr)
262     {
263         head_ = CreateNode();
264         tail_ = head_;
265
266         head_->next = NULL;
267         head_->prev = NULL;
268
269         head_->values[0] = value;
270         ++head_->count;
271
272         ++BListStats_.NodeCount;
273         ++BListStats_.ItemCount;
274
275         return;
276     }
277
278     if(head_->count == BListStats_.ArraySize)
279     {
280         BNode* newNode = CreateNode();
281
282         newNode->values[0] = value;
283         ++newNode->count;
284
285         newNode->prev = newNode;
286         newNode->next = head_;
287         head_ = newNode;
288
289         ++BListStats_.NodeCount;
290     }
291     else
292     {
293         for(int i = head_->count; i > 0; --i)
294         {
295             head_->values[i] = head_->values[i-1];
296         }
297
298         head_->values[0] = value;
299         ++head_->count;
300     }
301
302     ++BListStats_.ItemCount;
303 }
304
305 /******************************************************************************/
306 /*!
307 \fn     template<typename T, unsigned Size>
308         void BList<T, Size>::insert(const T& value)
309
310 \brief  Templated function to insert Node into a sorted list
311
312 \param  value - value of node to be inserted into the sorted list
313
314 */
315 /******************************************************************************/
316 template<typename T, unsigned Size>
317 void BList<T, Size>::insert(const T& value)
318 {
319     //if node is empty
320     if(head_ == NULL)
321     {
```

```
322         BNode* newNode = CreateNode();
323         ++BListStats_.NodeCount;
324 
325         newNode->values[0] = value;
326 
327         head_ = newNode;
328         tail_ = newNode;
329 
330         head_->next = NULL;
331         head_->prev = NULL;
332 
333         ++newNode->count;
334 
335         ++BListStats_.ItemCount;
336 
337         return;
338     }
339 
340     //if only 1 node in list and array size is greater than 1
341     if(head_ == tail_ && BListStats_.ArraySize != 1)
342     {
343         //if node is full, split node and reset pointer
344         if(head_->count == BListStats_.ArraySize)
345         {
346             head_ = SplitNode(head_, value);
347             tail_ = head_->next;
348         }
349 
350         //If there is room in the node, insert value into front, middle or back
351         //depend on how big the value is
352         else if(head_->count < BListStats_.ArraySize)
353         {
354             if(value < head_->values[0])
355             {
356                 InsertFront(head_, value);
357             }
358             else if(head_->values[head_->count -1] < value)
359             {
360                 InsertBack(head_, value);
361             }
362             else
363             {
364                 InsertMiddle(head_, value);
365             }
366 
367             ++head_->count;
368             ++BListStats_.ItemCount;
369         }
370 
371         return;
372     }
373 
374     BNode* NodePtr = head_;
375 
376     while(NodePtr)
377     {
378         unsigned count = NodePtr->count;
379 
380         //If array size == 1 and at end of list and first value < value
381         //make new node and place it at end of the list
382         if(BListStats_.ArraySize == 1 &&
383            NodePtr->next == NULL &&
384            NodePtr->values[0] < value)
385         {
386             BNode* newNode = CreateNode();
387             ++BListStats_.NodeCount;
388 
389             newNode->values[0] = value;
390             NodePtr->next = newNode;
391             NodePtr->prev = NULL;
392 
393             newNode->next = NULL;
394             newNode->prev = NodePtr;
395 
396             tail_ = newNode;
397             tail_->next = NULL;
398             tail_->prev = NodePtr;
399 
400             ++newNode->count;
401             ++BListStats_.ItemCount;
402 
403             return;
404         }
405         //If array size = 1, we are at head and value is less than first head
406         //value. Insert new node with value in front
407         else if (BListStats_.ArraySize == 1 &&
408                  NodePtr == head_ &&
409                  value < NodePtr->values[0])
410         {
411             BNode* newNode = CreateNode();
412             ++BListStats_.NodeCount;
413 
414             newNode->values[0] = value;
415             newNode->next = NodePtr;
416             newNode->prev = NULL;
417 
418             NodePtr->prev = newNode;
419 
420             head_ = newNode;
421             head_->prev = NULL;
422             head_->next = NodePtr;
423 
424             ++newNode->count;
425             ++BListStats_.ItemCount;
426 
427             return;
428         }
```

```
429         //If array size = 1 and value greater than first value and less than
430         //first value of next node, Insert new node in betwen these two
431         else if (BListStats_.ArraySize == 1 &&
432                  NodePtr->values[0] < value &&
433                  value < NodePtr->next->values[0])
434         {
435             BNode* newNode = CreateNode();
436             ++BListStats_.NodeCount;
437             newNode->values[0] = value;
438
439             BNode* temp = NodePtr->next;
440             NodePtr->next = newNode;
441
442             newNode->prev = NodePtr;
443             temp->prev = newNode;
444             newNode->next = temp;
445
446             ++newNode->count;
447             ++BListStats_.ItemCount;
448
449             return;
450         }
451
452         if(BListStats_.ArraySize != 1)
453         {
454             //If value < last value in node, insert value in the same node
455             //Check if next is null to prevent comparing with nodes that does
456             //not exist
457             if(value < NodePtr->values[count - 1] || NodePtr->next == NULL)
458             {
459                 if(NodePtr->next == NULL)
460                 {
461                     //if there is room in previous node and value is between
462                     //last node and start of first node,
463                     //place value in the last node
464                     if(NodePtr->prev != NULL &&
465                        NodePtr->prev->count < BListStats_.ArraySize &&
466                        NodePtr->prev->values[NodePtr->prev->count-1] < value &&
467                        value < NodePtr->values[0])
468                     {
469                         InsertBack(NodePtr->prev, value);
470
471                         ++NodePtr->prev->count;
472                         ++BListStats_.ItemCount;
473
474                         return;
475                     }
476
477                     //If node is full and previous node exist but also full and
478                     //value can fit between first and previous nodes, split the
479                     //previous node
480                     else if (NodePtr->prev != NULL &&
481                              NodePtr->count == BListStats_.ArraySize &&
482                              NodePtr->prev->count == BListStats_.ArraySize &&
483                              value < NodePtr->values[0] &&
484                              NodePtr->prev->values[NodePtr->prev->count - 1]<value)
485                     {
486                         NodePtr->prev = SplitNode(NodePtr->prev, value);
487                         return;
488                     }
489
490                     //If there is room in the current node, put the value in the
491                     //current node, insert into front, middle, back depending on
492                     //the value
493                     else if (count < BListStats_.ArraySize)
494                     {
495                         if(value < NodePtr->values[0])
496                         {
497                             InsertFront(NodePtr, value);
498                         }
499                         else if (NodePtr->values[count -1] < value)
500                         {
501                             InsertBack(NodePtr, value);
502                         }
503                         else
504                         {
505                             InsertMiddle(NodePtr, value);
506                         }
507
508                         ++NodePtr->count;
509                         ++BListStats_.ItemCount;
510
511                         return;
512                     }
513                     //If node is full, split the node
514                     else if (count == BListStats_.ArraySize)
515                     {
516                         NodePtr = SplitNode(NodePtr, value);
517                     }
518
519                     return;
520                 }
521
522                 //If node is full and previous node exist but is also full and
523                 //value can fit between first and previous node, split the
524                 //previous node
525                 else if(NodePtr->prev != NULL &&
526                         NodePtr->count == BListStats_.ArraySize &&
527                         NodePtr->prev->count == BListStats_.ArraySize &&
528                         value < NodePtr->values[0] &&
529                         NodePtr->prev->values[NodePtr->prev->count -1] < value)
530                 {
531                     NodePtr->prev = SplitNode(NodePtr->prev, value);
532                     return;
533                 }
534                 //If current node not full and previous node exist and there is
535                 //room in the previous node and value can fit between the two,
```

```
536                //
537                else if (NodePtr->prev != NULL &&
538                    NodePtr->count < BListStats_.ArraySize &&
539                    NodePtr->prev->count < BListStats_.ArraySize &&
540                    NodePtr->prev->values[NodePtr->prev->count - 1] < value
541                    &&value < NodePtr->values[0])
542                {
543                    InsertBack(NodePtr->prev, value);
544
545                    ++NodePtr->prev->count;
546                    ++BListStats_.ItemCount;
547                    return;
548                }
549                //current node is full and previous node exist and there is room
550                //in the previous node and value can fit between the two, insert
551                //into end of previous node
552                else if (NodePtr->count == BListStats_.ArraySize &&
553                    NodePtr->prev != NULL &&
554                    NodePtr->prev->count < BListStats_.ArraySize &&
555                    NodePtr->prev->values[NodePtr->prev->count -1] < value
556                    &&value < NodePtr ->values[0])
557                {
558                    InsertBack(NodePtr->prev, value);
559
560                    ++NodePtr->prev->count;
561                    ++BListStats_.ItemCount;
562
563                    return;
564                }
565                //If there is room in current node, put the value in the current
566                //node, insert into front, middle or end depending on the value
567                else if (count < BListStats_.ArraySize)
568                {
569                    if(value < NodePtr->values[0])
570                    {
571                        InsertFront(NodePtr, value);
572                    }
573                    else if (NodePtr->values[count -1] < value &&
574                        value < NodePtr->next->values[0])
575                    {
576                        InsertBack(NodePtr, value);
577                    }
578                    else
579                    {
580                        InsertMiddle(NodePtr, value);
581                    }
582
583                    ++NodePtr->count;
584                    ++BListStats_.ItemCount;
585
586                    return;
587                }
588                //if node is full, split the node
589                else if (count == BListStats_.ArraySize)
590                {
591                    NodePtr = SplitNode(NodePtr, value);
592                    return;
593                }
594            }
595        }
596
597        NodePtr = NodePtr->next;
598    }
599}
600
601/*****************************************************************************/
602/*!
603\fn     template<typename T, unsigned Size>
604        void BList<T, Size>::remove(int index)
605
606\brief  Templated function to remove node at a given index
607
608\param  index - the index to remove the node
609
610*/
611/*****************************************************************************/
612template<typename T, unsigned Size>
613void BList<T, Size>::remove(int index)
614{
615    BNode* NodePtr = head_;
616
617    int counter = NodePtr->count - 1;
618
619    //If negative index or index out of array count
620    if(index < 0 || index >= BListStats_.ItemCount)
621    {
622        throw(BListException(BListException::E_BAD_INDEX, "BAD_INDEX"));
623    }
624
625    //loop through until counter > index, then index is in that node
626    while(counter < index)
627    {
628        NodePtr = NodePtr->next;
629        counter += NodePtr->count;
630    }
631
632    //Minus count that was added to go to the beginning of node
633    counter -= NodePtr->count - 1;
634
635    //If array has only one values, dont shift values
636    if(BListStats_.ArraySize != 1)
637    {
638        for(unsigned i = index - counter; i < NodePtr->count; ++i)
639        {
640            NodePtr->values[i] = NodePtr->values[i+1];
641        }
642    }
```

```
643
644     --NodePtr->count;
645     --BListStats_.ItemCount;
646
647     NodePtr = head_;
648
649     //Remove head if count is 0, reset pointers
650     if(head_->count == 0)
651     {
652         BNode* temp = head_->next;
653         delete head_;
654
655         head_ = temp;
656         head_->prev = NULL;
657
658         --BListStats_.NodeCount;
659     }
660
661     //Remove tail if empty
662     if(tail_->count == 0)
663     {
664         BNode* temp = tail_->prev;
665         delete tail_;
666         tail_ = temp;
667         tail_->next = NULL;
668
669         --BListStats_.NodeCount;
670     }
671
672     NodePtr = head_;
673
674     //Loop through to find if other node are empty
675     while(NodePtr)
676     {
677         if(NodePtr->count == 0)
678         {
679             BNode* temp = NodePtr->next;
680             BNode* tempPrevious = NodePtr->prev;
681
682             delete NodePtr;
683             NodePtr = temp;
684             NodePtr->prev = tempPrevious;
685             tempPrevious->next = NodePtr;
686
687             --BListStats_.NodeCount;
688         }
689
690         NodePtr = NodePtr->next;
691     }
692 }
693
694 /*****************************************************************************/
695 /*!
696 \fn     template<typename T, unsigned Size>
697         void BList<T, Size>::remove_by_value(const T& value)
698
699 \brief  Templated Function to remove node of specific value
700
701 \param  value - value of node to be removed
702
703 */
704 /*****************************************************************************/
705 template<typename T, unsigned Size>
706 void BList<T, Size>::remove_by_value(const T& value)
707 {
708     BNode* NodePtr = head_;
709
710     while(NodePtr)
711     {
712         for(unsigned i = 0; i < NodePtr->count; ++i)
713         {
714             //If value is found
715             if(NodePtr->values[i] == value && BListStats_.ArraySize != 1)
716             {
717                 //Shift all values down
718                 for(unsigned j = i; j < NodePtr->count; ++j)
719                 {
720                     NodePtr->values[j] = NodePtr->values[j+1];
721                 }
722
723                 --NodePtr->count;
724                 --BListStats_.ItemCount;
725             }
726
727             if(NodePtr->values[i] == value && BListStats_.ArraySize == 1)
728             {
729                 --NodePtr->count;
730                 --BListStats_.ItemCount;
731             }
732         }
733
734         NodePtr = NodePtr->next;
735     }
736
737     //Remove head if count = 0, reset pointers
738     if(head_->count == 0)
739     {
740         BNode* temp = head_->next;
741         delete head_;
742
743         head_ = temp;
744         head_->prev = NULL;
745         --BListStats_.NodeCount;
746     }
747
748     //Remove tail if empty
749     if(tail_->count == 0)
```

```
    {
        BNode* temp = tail_->prev;
        delete tail_;

        tail_ = temp;
        tail_->next = NULL;
        --BListStats_.NodeCount;
    }

    NodePtr = head_;

    //Loop through list to find if other nodes are empty
    while(NodePtr)
    {
        if(NodePtr->count == 0)
        {
            BNode* temp = NodePtr->next;
            BNode* tempPrevious = NodePtr->prev;

            delete NodePtr;

            NodePtr = temp;
            NodePtr->prev = tempPrevious;
            tempPrevious->next = NodePtr;

            --BListStats_.NodeCount;
        }

        NodePtr = NodePtr->next;
    }
}

/*****************************************************************************/
/*!
\fn     template<typename T, unsigned Size>
        int BList<T, Size>::find(const T& value) const

\brief  Templated Function to find node of specific value

\param  value - value of node to find.

*/
/*****************************************************************************/
template<typename T, unsigned Size>
int BList<T, Size>::find(const T& value) const
{
    (void)value;

    //If list is unsorted
    if(sort == false)
    {
        if(head_ == NULL || head_->values[0] == value)
            return 0;

        int count = 0;

        BNode* NodePtr = head_;

        while(NodePtr)
        {
            for(unsigned i = 0; i < NodePtr->count; ++i)
            {
                ++count;

                //If value found
                if(NodePtr->values[i] == value)
                    return count - 1;
            }

            NodePtr = NodePtr->next;
        }

        //if value not found
        return -1;
    }
    //Perform binary search if list is sorted
    else
    {
        if(head_ == NULL || head_->values[0] == value)
            return 0;

        int lower = 0;
        int upper = BListStats_.ItemCount - 1;
        int position = (lower + upper) /2;

        while(!((*this)[position] == value) && (lower < upper))
        {
            //if value < middle
            if(value < (*this)[position])
            {
                upper = position - 1;
            }
            //if value > middle
            else
            {
                lower = position + 1;
            }

            //find new middle position
            position = (lower + upper) / 2;

        }

        //value not found
        if(lower > upper)
            return -1;
        //value found
```

```
857         else
858             return position;
859     }
860     return -1;
861 }
862 
863 /*****************************************************************************/
864 /*!
865 \fn     template<typename T, unsigned Size>
866         T& BList<T, Size>::operator[](int index)
867 
868 \brief  Templated Function to find value at a given index
869 
870 \param  index - index of node to be found
871 
872 \return Returns value of the node at the current index
873 */
874 /*****************************************************************************/
875 template<typename T, unsigned Size>
876 T& BList<T, Size>::operator[](int index)
877 {
878     BNode* NodePtr = head_;
879 
880     int counter = 0;
881 
882     //If bad index given
883     if(index < 0 || index >= BListStats_.ItemCount)
884     {
885         throw(BListException(BListException::E_BAD_INDEX,"BAD INDEX"));
886     }
887 
888     while(NodePtr)
889     {
890         //add current node count to counter
891         counter += NodePtr->count;
892 
893         //counter > index, index in current node
894         if(counter > index)
895         {
896             //if node round, minus what added to start at beginning of node
897             counter -= NodePtr->count;
898             return NodePtr->values[index - counter];
899         }
900 
901         NodePtr = NodePtr->next;
902     }
903 
904     throw (BListException(BListException::E_BAD_INDEX, "BAD INDEX"));
905     return NodePtr->values[0];
906 }
907 
908 /*****************************************************************************/
909 /*!
910 \fn     template<typename T, unsigned Size>
911         T& BList<T, Size>::operator[](int index) const
912 
913 \brief  Templated Function to find value at a given index
914 
915 \param  index - index of node to be found
916 
917 \return Returns value of the node at the current index
918 */
919 /*****************************************************************************/
920 template<typename T, unsigned Size>
921 const T& BList<T, Size>::operator[](int index)const
922 {
923   BNode* NodePtr = head_;
924 
925     int counter = 0;
926 
927     //If bad index given
928     if(index < 0 || index >= BListStats_.ItemCount)
929     {
930         throw(BListException(BListException::E_BAD_INDEX,"BAD INDEX"));
931     }
932 
933     while(NodePtr)
934     {
935         //Add current node count to counter
936         counter += NodePtr->count;
937 
938         //If count > index, index in current node
939         if(counter > index)
940         {
941             //if node round, minus what added to start at beginning of node
942             counter -= NodePtr->count;
943             return NodePtr->values[index - counter];
944         }
945 
946         NodePtr = NodePtr->next;
947     }
948 
949     throw (BListException(BListException::E_BAD_INDEX, "BAD INDEX"));
950     return NodePtr->values[0];
951 }
952 
953 /*****************************************************************************/
954 /*!
955 \fn     template<typename T, unsigned Size>
956         void BList<T, Size>::clear()
957 
958 \brief  Templated Function to delete all the nodes in the list
959 
960 */
961 /*****************************************************************************/
962 template<typename T, unsigned Size>
963 void BList<T, Size>::clear()
```

```
964  {
965      //if head empty, nothing to clear
966      if(head_ == NULL)
967          return;
968
969      while(head_)
970      {
971          //If one node in list
972          if(head_ == tail_)
973          {
974              delete head_;
975              break;
976          }
977
978          //delete all node
979          BNode* temp = head_->next;
980          //delete current node
981          delete head_;
982          //reset pointers
983          head_ = temp;
984          head_->prev = NULL;
985
986          head_ = head_->next;
987      }
988
989      BListStats_.ItemCount = 0;
990      BListStats_.NodeCount = 0;
991
992      head_ = NULL;
993      tail_ = NULL;
994      sort = true;
995  }
996
997  /*****************************************************************************/
998  /*!
999  \fn     template<typename T, unsigned Size>
1000         BListStats BList<T, Size>::GetStats() const
1001
1002 \brief  Templated Function to return BListStats struct
1003
1004 \return Returns current BListStats_
1005 */
1006 /*****************************************************************************/
1007 template<typename T, unsigned Size>
1008 BListStats BList<T, Size>::GetStats() const
1009 {
1010     return BListStats_;
1011 }
1012
1013 template<typename T, unsigned Size>
1014 size_t BList<T, Size>::size() const
1015 {
1016     return BListStats_.ItemCount;
1017 }
1018
1019 template<typename T, unsigned Size>
1020 typename BList<T, Size>::BNode* BList<T, Size>::CreateNode()
1021 {
1022     BNode* newNode = 0;
1023
1024     try
1025     {
1026         newNode = new BNode;
1027     }
1028     catch(const std::exception& e)
1029     {
1030         throw(BListException(BListException::E_NO_MEMORY, e.what()));
1031     }
1032
1033     return newNode;
1034 }
1035
1036 /*****************************************************************************/
1037 /*!
1038 \fn     template<typename T, unsigned Size>
1039         typename BList<T, Size>::BNode* BList<T, Size>::SplitNode(BNode* node,
1040         const T value)
1041
1042 \brief  Templated Function to split nodes and put value into the right position
1043
1044 \param  node - Head of the node to be split
1045 \param  value - value of node to be inserted into the sorted list
1046
1047 \return Returns pointer to the first node that is split
1048 */
1049 /*****************************************************************************/
1050 template<typename T, unsigned Size>
1051 typename BList<T, Size>::BNode* BList<T, Size>::SplitNode(BNode* node,
1052 const T value)
1053 {
1054     //Make 2 new nodes to split them
1055     BNode* FirstNode = CreateNode();
1056     BNode* SecondNode = CreateNode();
1057
1058     ++BListStats_.NodeCount;
1059
1060     unsigned j = 0;
1061
1062     //Place first half of data into first node
1063     for(unsigned i = 0; i < (node->count)/2; i++)
1064     {
1065         FirstNode->values[j] = node->values[i];
1066         ++FirstNode->count;
1067         j++;
1068     }
1069
1070     j = 0;
```

```
1071
1072        //Place second half of data into second node
1073        for(unsigned i = (node->count)/2; i < node->count; i++)
1074        {
1075            SecondNode->values[j] = node->values[i];
1076            ++SecondNode->count;
1077            j++;
1078        }
1079
1080        //Insert value into one of the two split nodes
1081        //If value < all the value in second node, it's in the first node
1082        //If not, put in second node
1083        if(value < SecondNode->values[0])
1084        {
1085            if(value < FirstNode->values[0])
1086            {
1087                InsertFront(FirstNode, value);
1088            }
1089            //if value is < first node last value and value < secondNode first value
1090            else if(FirstNode->values[FirstNode->count - 1] < value &&
1091                    value < SecondNode->values[0])
1092            {
1093                InsertBack(FirstNode, value);
1094            }
1095            else
1096            {
1097                InsertMiddle(FirstNode, value);
1098            }
1099
1100            ++FirstNode->count;
1101        }
1102        //Insert into second node
1103        else
1104        {
1105            if(value < SecondNode->values[0])
1106            {
1107                InsertFront(SecondNode, value);
1108            }
1109            else if (SecondNode->values[SecondNode->count - 1] < value)
1110            {
1111                InsertBack(SecondNode, value);
1112            }
1113            else
1114            {
1115                InsertMiddle(SecondNode, value);
1116            }
1117
1118            ++SecondNode->count;
1119        }
1120
1121        ++BListStats_.ItemCount;
1122
1123        BNode* prev = node->prev;
1124        BNode* next = node->next;
1125
1126        //If nothing behind or in front of given node
1127        if(prev == NULL && next == NULL)
1128        {
1129            FirstNode->next = SecondNode;
1130            SecondNode->prev = FirstNode;
1131            FirstNode->prev = NULL;
1132            SecondNode->prev = FirstNode;
1133        }
1134        //If nothing behind the given node
1135        else if (prev == NULL && next != NULL)
1136        {
1137            FirstNode->next = SecondNode;
1138            SecondNode->prev = FirstNode;
1139            FirstNode->prev = NULL;
1140            SecondNode->next = next;
1141            next->prev = SecondNode;
1142        }
1143        //If nothing in front of the given node
1144        else if(prev != NULL && next == NULL)
1145        {
1146            FirstNode->next = SecondNode;
1147            SecondNode->prev = FirstNode;
1148            prev->next = FirstNode;
1149            FirstNode->prev = prev;
1150            SecondNode->next = NULL;
1151        }
1152        //If there is nodes pointing to and from the given nodes
1153        else if( prev != NULL && next != NULL)
1154        {
1155            FirstNode->next = SecondNode;
1156            SecondNode->prev = FirstNode;
1157            prev->next = FirstNode;
1158            FirstNode->prev = prev;
1159            SecondNode->next = next;
1160            next->prev = SecondNode;
1161        }
1162
1163        //Reset head and tail if given node is a head or tail
1164        if(node == head_)
1165            head_ = FirstNode;
1166
1167        if(node == tail_)
1168            tail_ = SecondNode;
1169
1170        delete node;
1171        return FirstNode;
1172    }
1173
1174    /******************************************************************************/
1175    /*!
1176    \fn     template<typename T, unsigned Size>
1177            void BList<T, Size>::InsertFront(BNode* node, const T value)
```

```
1178  \brief  Templated Function to insert value to the front of node
1179
1180  \param  node - Head of the node to be inserted
1181  \param  value - value of node to be inserted into the sorted list
1182
1183  */
1184  /*****************************************************************************/
1185  template<typename T, unsigned Size>
1186  void BList<T, Size>::InsertFront(BNode* node, const T value)
1187  {
1188      //Shift value to right one by one and insert at the front
1189      for(unsigned i = node->count; i > 0; --i)
1190          node->values[i] = node->values[i-1];
1191
1192      node->values[0] = value;
1193  }
1194
1195  /*****************************************************************************/
1196  /*!
1197  \fn     template<typename T, unsigned Size>
1198          void BList<T, Size>::InsertBack(BNode* node, const T value)
1199
1200  \brief  Templated Function to insert value to the back of node
1201
1202  \param  node - Head of the node to be inserted
1203  \param  value - value of node to be inserted into the sorted list
1204
1205  */
1206  /*****************************************************************************/
1207  template<typename T, unsigned Size>
1208  void BList<T, Size>::InsertBack(BNode* node, const T value)
1209  {
1210      //Insert value at end of array
1211      node->values[node->count] = value;
1212  }
1213
1214  /*****************************************************************************/
1215  /*!
1216  \fn     template<typename T, unsigned Size>
1217          void BList<T, Size>::InsertMiddle(BNode* node, const T value)
1218
1219  \brief  Templated Function to insert value to the middle of node
1220
1221  \param  node - Head of the node to be inserted
1222  \param  value - value of node to be inserted into the sorted list
1223
1224  */
1225  /*****************************************************************************/
1226  template<typename T, unsigned Size>
1227  void BList<T, Size>::InsertMiddle(BNode* node, const T value)
1228  {
1229      for(unsigned i = 0; i < node->count; ++i)
1230      {
1231          //Find where should the values be
1232          if(node->values[i] < value && value < node->values[i+1])
1233          {
```

VPL

◄ Assignment 1: Object Allocator

Jump to...

Introduction to Data Structures ►