cs280s21-b.sg

Dashboard / My courses / cs280s21-b.sg / General / Assignment 2: B List

Description Submission Edit Submission view

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

```
////////////////////////////////////////////////////////////////////////////////
#ifndef BLIST_H
#define BLIST_H
////////////////////////////////////////////////////////////////////////////////

#include <string> // error strings

/*!
  The exception class for BList
*/
class BListException : public std::exception
{
  private:
    int m_ErrCode;             //!< One of E_NO_MEMORY, E_BAD_INDEX,E_DATA_ERROR
    std::string m_Description; //!< Description of the exception

  public:
    /*!
      Constructor

      \param ErrCode
        The error code for the exception.

      \param Description
        The description of the exception.
    */
    BListException(int ErrCode, const std::string& Description) :
    m_ErrCode(ErrCode), m_Description(Description) {};

    /*!
      Get the kind of exception

      \return
        One of E_NO_MEMORY, E_BAD_INDEX, E_DATA_ERROR
    */
    virtual int code() const {
      return m_ErrCode;
    }

    /*!
      Get the human-readable text for the exception

      \return
        The description of the exception
    */
    virtual const char *what() const throw() {
      return m_Description.c_str();
    }

    /*!
      Destructor is "implemented" because it needs to be virtual
    */
    virtual ~BListException() {
    }

    //! The reason for the exception
    enum BLIST_EXCEPTION {E_NO_MEMORY, E_BAD_INDEX, E_DATA_ERROR};
};

/*!
  Statistics about the BList
*/
struct BListStats
{
    //!< Default constructor
  BListStats() : NodeSize(0), NodeCount(0), ArraySize(0), ItemCount(0)  {};

  /*!
    Non-default constructor

    \param nsize
      Size of the node

    \param ncount
      Number of nodes in the list

    \param asize
      Number of elements in each node (array)

    \param count
      Number of items in the list

  */
  BListStats(size_t nsize, int ncount, int asize, int count) :
  NodeSize(nsize), NodeCount(ncount), ArraySize(asize), ItemCount(count)  {};

  size_t NodeSize; //!< Size of a node (via sizeof)
  int NodeCount;   //!< Number of nodes in the list
  size_t ArraySize;   //!< Max number of items in each node
  int ItemCount;   //!< Number of items in the entire list
};

/*!
  The BList class
*/
template <typename T, unsigned Size = 1>
class BList
{

  public:
    /*!
      Node struct for the BList
    */
    struct BNode
    {
      BNode *next;    //!< pointer to next BNode
      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

```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;
    }

    //Make nodeHead point be tail
    BNode* nodeHead = tail_;

    if(nodeHead->count == BListStats_.ArraySize)
    {
        BNode* newNode = CreateNode();

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

        //Reset pointers, make this node the tail
        tail_->next = newNode;
        newNode->prev = tail_;
        newNode->next = NULL;
        tail_ = newNode;

        ++BListStats_.NodeCount;
    }
    else
    {
        nodeHead->values[nodeHead->count] = value;
        ++nodeHead->count;
    }

    ++BListStats_.ItemCount;
}

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

\brief  Templated function to insert nodes from the front of the list

\param  value - value of node to be inserted

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

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

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

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

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

        return;
    }

    if(head_->count == BListStats_.ArraySize)
    {
        BNode* newNode = CreateNode();

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

        newNode->prev = newNode;
        newNode->next = head_;
        head_ = newNode;

        ++BListStats_.NodeCount;
    }
    else
    {
        for(int i = head_->count; i > 0; --i)
        {
            head_->values[i] = head_->values[i-1];
        }

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

    ++BListStats_.ItemCount;
}

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

\brief  Templated function to insert Node into a sorted list

\param  value - value of node to be inserted into the sorted list

*/
/******************************************************************************/
template<typename T, unsigned Size>
void BList<T, Size>::insert(const T& value)
{
    //if node is empty
    if(head_ == NULL)
    {
        BNode* newNode = CreateNode();
        ++BListStats_.NodeCount;

        newNode->values[0] = value;

        head_ = newNode;
        tail_ = newNode;

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

        ++newNode->count;

        ++BListStats_.ItemCount;

        return;
    }

    //if only 1 node in list and array size is greater than 1
    if(head_ == tail_ && BListStats_.ArraySize != 1)
    {
        //if node is full, split node and reset pointer
        if(head_->count == BListStats_.ArraySize)
        {
            head_ = SplitNode(head_, value);
            tail_ = head_->next;
        }

        //If there is room in the node, insert value into front, middle or back
        //depend on how big the value is
        else if(head_->count < BListStats_.ArraySize)
        {
            if(value < head_->values[0])
            {
                InsertFront(head_, value);
            }
            else if(head_->values[head_->count -1] < value)
            {
                InsertBack(head_, value);
            }
            else
            {
                InsertMiddle(head_, value);
            }

            ++head_->count;
            ++BListStats_.ItemCount;
        }

        return;
    }

    BNode* NodePtr = head_;

    while(NodePtr)
    {
        unsigned count = NodePtr->count;

        //If array size == 1 and at end of list and first value < value
        //make new node and place it at end of the list
        if(BListStats_.ArraySize == 1 &&
           NodePtr->next == NULL &&
           NodePtr->values[0] < value)
        {
            BNode* newNode = CreateNode();
            ++BListStats_.NodeCount;

            newNode->values[0] = value;
            NodePtr->next = newNode;
            NodePtr->prev = NULL;

            newNode->next = NULL;
            newNode->prev = NodePtr;

            tail_ = newNode;
            tail_->next = NULL;
            tail_->prev = NodePtr;

            ++newNode->count;
            ++BListStats_.ItemCount;

            return;
        }
        //If array size = 1, we are at head and value is less than first head
        //value. Insert new node with value in front
        else if (BListStats_.ArraySize == 1 &&
                 NodePtr == head_ &&
                 value < NodePtr->values[0])
        {
            BNode* newNode = CreateNode();
            ++BListStats_.NodeCount;

            newNode->values[0] = value;
            newNode->next = NodePtr;
            newNode->prev = NULL;

            NodePtr->prev = newNode;

            head_ = newNode;
            head_->prev = NULL;
            head_->next = NodePtr;

            ++newNode->count;
            ++BListStats_.ItemCount;

            return;
        }
        //If array size = 1 and value greater than first value and less than
        //first value of next node, Insert new node in betwen these two
        else if (BListStats_.ArraySize == 1 &&
                 NodePtr->values[0] < value &&
                 value < NodePtr->next->values[0])
        {
            BNode* newNode = CreateNode();
            ++BListStats_.NodeCount;
            newNode->values[0] = value;

            BNode* temp = NodePtr->next;
            NodePtr->next = newNode;

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

            ++newNode->count;
            ++BListStats_.ItemCount;

            return;
        }

        if(BListStats_.ArraySize != 1)
        {
            //If value < last value in node, insert value in the same node
            //Check if next is null to prevent comparing with nodes that does
            //not exist
            if(value < NodePtr->values[count - 1] || NodePtr->next == NULL)
            {
                if(NodePtr->next == NULL)
                {
                    //if there is room in previous node and value is between
                    //last node and start of first node,
                    //place value in the last node
                    if(NodePtr->prev != NULL &&
                       NodePtr->prev->count < BListStats_.ArraySize &&
                       NodePtr->prev->values[NodePtr->prev->count-1] < value &&
                       value < NodePtr->values[0])
                    {
                        InsertBack(NodePtr->prev, value);

                        ++NodePtr->prev->count;
                        ++BListStats_.ItemCount;

                        return;
                    }

                    //If node is full and previous node exist but also full and
                    //value can fit between first and previous nodes, split the
                    //previous node
                    else if (NodePtr->prev != NULL &&
                         NodePtr->count == BListStats_.ArraySize &&
                         NodePtr->prev->count == BListStats_.ArraySize &&
                         value < NodePtr->values[0] &&
                         NodePtr->prev->values[NodePtr->prev->count - 1]<value)
                    {
                        NodePtr->prev = SplitNode(NodePtr->prev, value);
                        return;
                    }

                    //If there is room in the current node, put the value in the
                    //current node, insert into front, middle, back depending on
                    //the value
                    else if (count < BListStats_.ArraySize)
                    {
                        if(value < NodePtr->values[0])
                        {
                            InsertFront(NodePtr, value);
                        }
                        else if (NodePtr->values[count -1] < value)
                        {
                            InsertBack(NodePtr, value);
                        }
                        else
                        {
                            InsertMiddle(NodePtr, value);
                        }

                        ++NodePtr->count;
                        ++BListStats_.ItemCount;

                        return;
                    }
                    //If node is full, split the node
                    else if (count == BListStats_.ArraySize)
                    {
                        NodePtr = SplitNode(NodePtr, value);
                    }

                    return;
                }

                //If node is full and previous node exist but is also full and
                //value can fit between first and previous node, split the
                //previous node
                else if(NodePtr->prev != NULL &&
                        NodePtr->count == BListStats_.ArraySize &&
                        NodePtr->prev->count == BListStats_.ArraySize &&
                        value < NodePtr->values[0] &&
                        NodePtr->prev->values[NodePtr->prev->count -1] < value)
                {
                    NodePtr->prev = SplitNode(NodePtr->prev, value);
                    return;
                }
                //If current node not full and previous node exist and there is
                //room in the previous node and value can fit between the two,
                //
                else if (NodePtr->prev != NULL &&
                        NodePtr->count < BListStats_.ArraySize &&
                        NodePtr->prev->count < BListStats_.ArraySize &&
                        NodePtr->prev->values[NodePtr->prev->count - 1] < value
                        &&value < NodePtr->values[0])
                {
                    InsertBack(NodePtr->prev, value);

                    ++NodePtr->prev->count;
                    ++BListStats_.ItemCount;
                    return;
                }
                //current node is full and previous node exist and there is room
                //in the previous node and value can fit between the two, insert
                //into end of previous node
                else if (NodePtr->count == BListStats_.ArraySize &&
                        NodePtr->prev != NULL &&
                        NodePtr->prev->count < BListStats_.ArraySize &&
                        NodePtr->prev->values[NodePtr->prev->count -1] < value
                        &&value < NodePtr ->values[0])
                {
                    InsertBack(NodePtr->prev, value);

                    ++NodePtr->prev->count;
                    ++BListStats_.ItemCount;

                    return;
                }
                //If there is room in current node, put the value in the current
                //node, insert into front, middle or end depending on the value
                else if (count < BListStats_.ArraySize)
                {
                    if(value < NodePtr->values[0])
                    {
                        InsertFront(NodePtr, value);
                    }
                    else if (NodePtr->values[count -1] < value &&
                            value < NodePtr->next->values[0])
                    {
                        InsertBack(NodePtr, value);
                    }
                    else
                    {
                        InsertMiddle(NodePtr, value);
                    }

                    ++NodePtr->count;
                    ++BListStats_.ItemCount;

                    return;
                }
                //if node is full, split the node
                else if (count == BListStats_.ArraySize)
                {
                    NodePtr = SplitNode(NodePtr, value);
                    return;
                }
            }
        }

        NodePtr = NodePtr->next;
    }
}

/******************************************************************************/
/*!
\fn     template<typename T, unsigned Size>
        void BList<T, Size>::remove(int index)

\brief  Templated function to remove node at a given index

\param  index - the index to remove the node

*/
/******************************************************************************/
template<typename T, unsigned Size>
void BList<T, Size>::remove(int index)
{
    BNode* NodePtr = head_;

    int counter = NodePtr->count - 1;

    //If negative index or index out of array count
    if(index < 0 || index >= BListStats_.ItemCount)
    {
        throw(BListException(BListException::E_BAD_INDEX, "BAD_INDEX"));
    }

    //loop through until counter > index, then index is in that node
    while(counter < index)
    {
        NodePtr = NodePtr->next;
        counter += NodePtr->count;
    }

    //Minus count that was added to go to the beginning of node
    counter -= NodePtr->count - 1;

    //If array has only one values, dont shift values
    if(BListStats_.ArraySize != 1)
    {
        for(unsigned i = index - counter; i < NodePtr->count; ++i)
        {
            NodePtr->values[i] = NodePtr->values[i+1];
        }
    }

    --NodePtr->count;
    --BListStats_.ItemCount;

    NodePtr = head_;

    //Remove head if count is 0, reset pointers
    if(head_->count == 0)
    {
        BNode* temp = head_->next;
        delete head_;

        head_ = temp;
        head_->prev = NULL;

        --BListStats_.NodeCount;
    }

    //Remove tail if empty
    if(tail_->count == 0)
    {
        BNode* temp = tail_->prev;
        delete tail_;
        tail_ = temp;
        tail_->next = NULL;

        --BListStats_.NodeCount;
    }

    NodePtr = head_;

    //Loop through to find if other node 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>
        void BList<T, Size>::remove_by_value(const T& value)

\brief  Templated Function to remove node of specific value

\param  value - value of node to be removed

*/
/******************************************************************************/
template<typename T, unsigned Size>
void BList<T, Size>::remove_by_value(const T& value)
{
    BNode* NodePtr = head_;

    while(NodePtr)
    {
        for(unsigned i = 0; i < NodePtr->count; ++i)
        {
            //If value is found
            if(NodePtr->values[i] == value && BListStats_.ArraySize != 1)
            {
                //Shift all values down
                for(unsigned j = i; j < NodePtr->count; ++j)
                {
                    NodePtr->values[j] = NodePtr->values[j+1];
                }

                --NodePtr->count;
                --BListStats_.ItemCount;
            }

            if(NodePtr->values[i] == value && BListStats_.ArraySize == 1)
            {
                --NodePtr->count;
                --BListStats_.ItemCount;
            }
        }

        NodePtr = NodePtr->next;
    }

    //Remove head if count = 0, reset pointers
    if(head_->count == 0)
    {
        BNode* temp = head_->next;
        delete head_;

        head_ = temp;
        head_->prev = NULL;
        --BListStats_.NodeCount;
    }

    //Remove tail if empty
    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
        else
            return position;
    }
    return -1;
}

/******************************************************************************/
/*!
\fn     template<typename T, unsigned Size>
        T& BList<T, Size>::operator[](int index)

\brief  Templated Function to find value at a given index

\param  index - index of node to be found

\return Returns value of the node at the current index
*/
/******************************************************************************/
template<typename T, unsigned Size>
T& BList<T, Size>::operator[](int index)
{
    BNode* NodePtr = head_;

    int counter = 0;

    //If bad index given
    if(index < 0 || index >= BListStats_.ItemCount)
    {
        throw(BListException(BListException::E_BAD_INDEX,"BAD INDEX"));
    }

    while(NodePtr)
    {
        //add current node count to counter
        counter += NodePtr->count;

        //counter > index, index in current node
        if(counter > index)
        {
            //if node round, minus what added to start at beginning of node
            counter -= NodePtr->count;
            return NodePtr->values[index - counter];
        }

        NodePtr = NodePtr->next;
    }

    throw (BListException(BListException::E_BAD_INDEX, "BAD INDEX"));
    return NodePtr->values[0];
}

/******************************************************************************/
/*!
\fn     template<typename T, unsigned Size>
        T& BList<T, Size>::operator[](int index) const

\brief  Templated Function to find value at a given index

\param  index - index of node to be found

\return Returns value of the node at the current index
*/
/******************************************************************************/
template<typename T, unsigned Size>
const T& BList<T, Size>::operator[](int index)const
{
  BNode* NodePtr = head_;

    int counter = 0;

    //If bad index given
    if(index < 0 || index >= BListStats_.ItemCount)
    {
        throw(BListException(BListException::E_BAD_INDEX,"BAD INDEX"));
    }

    while(NodePtr)
    {
        //Add current node count to counter
        counter += NodePtr->count;

        //If count > index, index in current node
        if(counter > index)
        {
            //if node round, minus what added to start at beginning of node
            counter -= NodePtr->count;
            return NodePtr->values[index - counter];
        }

        NodePtr = NodePtr->next;
    }

    throw (BListException(BListException::E_BAD_INDEX, "BAD INDEX"));
    return NodePtr->values[0];
}

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

\brief  Templated Function to delete all the nodes in the list

*/
/******************************************************************************/
template<typename T, unsigned Size>
void BList<T, Size>::clear()
```

```
{
    //if head empty, nothing to clear
    if(head_ == NULL)
        return;

    while(head_)
    {
        //If one node in list
        if(head_ == tail_)
        {
            delete head_;
            break;
        }

        //delete all node
        BNode* temp = head_->next;
        //delete current node
        delete head_;
        //reset pointers
        head_ = temp;
        head_->prev = NULL;

        head_ = head_->next;
    }

    BListStats_.ItemCount = 0;
    BListStats_.NodeCount = 0;

    head_ = NULL;
    tail_ = NULL;
    sort = true;
}

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

\brief  Templated Function to return BListStats struct

\return Returns current BListStats_
*/
/******************************************************************************/
template<typename T, unsigned Size>
BListStats BList<T, Size>::GetStats() const
{
    return BListStats_;
}

template<typename T, unsigned Size>
size_t BList<T, Size>::size() const
{
    return BListStats_.ItemCount;
}

template<typename T, unsigned Size>
typename BList<T, Size>::BNode* BList<T, Size>::CreateNode()
{
    BNode* newNode = 0;

    try
    {
        newNode = new BNode;
    }
    catch(const std::exception& e)
    {
        throw(BListException(BListException::E_NO_MEMORY, e.what()));
    }

    return newNode;
}

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

\brief  Templated Function to split nodes and put value into the right position

\param  node - Head of the node to be split
\param  value - value of node to be inserted into the sorted list

\return Returns pointer to the first node that is split
*/
/******************************************************************************/
template<typename T, unsigned Size>
typename BList<T, Size>::BNode* BList<T, Size>::SplitNode(BNode* node,
const T value)
{
    //Make 2 new nodes to split them
    BNode* FirstNode = CreateNode();
    BNode* SecondNode = CreateNode();

    ++BListStats_.NodeCount;

    unsigned j = 0;

    //Place first half of data into first node
    for(unsigned i = 0; i < (node->count)/2; i++)
    {
        FirstNode->values[j] = node->values[i];
        ++FirstNode->count;
        j++;
    }

    j = 0;
```

```cpp

    //Place second half of data into second node
    for(unsigned i = (node->count)/2; i < node->count; i++)
    {
        SecondNode->values[j] = node->values[i];
        ++SecondNode->count;
        j++;
    }

    //Insert value into one of the two split nodes
    //If value < all the value in second node, it's in the first node
    //If not, put in second node
    if(value < SecondNode->values[0])
    {
        if(value < FirstNode->values[0])
        {
            InsertFront(FirstNode, value);
        }
        //if value is < first node last value and value < secondNode first value
        else if(FirstNode->values[FirstNode->count - 1] < value &&
                value < SecondNode->values[0])
        {
            InsertBack(FirstNode, value);
        }
        else
        {
            InsertMiddle(FirstNode, value);
        }

        ++FirstNode->count;
    }
    //Insert into second node
    else
    {
        if(value < SecondNode->values[0])
        {
            InsertFront(SecondNode, value);
        }
        else if (SecondNode->values[SecondNode->count - 1] < value)
        {
            InsertBack(SecondNode, value);
        }
        else
        {
            InsertMiddle(SecondNode, value);
        }

        ++SecondNode->count;
    }

    ++BListStats_.ItemCount;

    BNode* prev = node->prev;
    BNode* next = node->next;

    //If nothing behind or in front of given node
    if(prev == NULL && next == NULL)
    {
        FirstNode->next = SecondNode;
        SecondNode->prev = FirstNode;
        FirstNode->prev = NULL;
        SecondNode->prev = FirstNode;
    }
    //If nothing behind the given node
    else if (prev == NULL && next != NULL)
    {
        FirstNode->next = SecondNode;
        SecondNode->prev = FirstNode;
        FirstNode->prev = NULL;
        SecondNode->next = next;
        next->prev = SecondNode;
    }
    //If nothing in front of the given node
    else if(prev != NULL && next == NULL)
    {
        FirstNode->next = SecondNode;
        SecondNode->prev = FirstNode;
        prev->next = FirstNode;
        FirstNode->prev = prev;
        SecondNode->next = NULL;
    }
    //If there is nodes pointing to and from the given nodes
    else if( prev != NULL && next != NULL)
    {
        FirstNode->next = SecondNode;
        SecondNode->prev = FirstNode;
        prev->next = FirstNode;
        FirstNode->prev = prev;
        SecondNode->next = next;
        next->prev = SecondNode;
    }

    //Reset head and tail if given node is a head or tail
    if(node == head_)
        head_ = FirstNode;

    if(node == tail_)
        tail_ = SecondNode;

    delete node;
    return FirstNode;
}

/******************************************************************************/
/*!
\fn     template<typename T, unsigned Size>
        void BList<T, Size>::InsertFront(BNode* node, const T value)
```

```
\brief  Templated Function to insert value to the front of node

\param  node - Head of the node to be inserted
\param  value - value of node to be inserted into the sorted list

*/
/******************************************************************************/
template<typename T, unsigned Size>
void BList<T, Size>::InsertFront(BNode* node, const T value)
{
    //Shift value to right one by one and insert at the front
    for(unsigned i = node->count; i > 0; --i)
        node->values[i] = node->values[i-1];

    node->values[0] = value;
}

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

\brief  Templated Function to insert value to the back of node

\param  node - Head of the node to be inserted
\param  value - value of node to be inserted into the sorted list

*/
/******************************************************************************/
template<typename T, unsigned Size>
void BList<T, Size>::InsertBack(BNode* node, const T value)
{
    //Insert value at end of array
    node->values[node->count] = value;
}

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

\brief  Templated Function to insert value to the middle of node

\param  node - Head of the node to be inserted
\param  value - value of node to be inserted into the sorted list

*/
/******************************************************************************/
template<typename T, unsigned Size>
void BList<T, Size>::InsertMiddle(BNode* node, const T value)
{
    for(unsigned i = 0; i < node->count; ++i)
    {
        //Find where should the values be
        if(node->values[i] < value && value < node->values[i+1])
        {
```

VPL

◄ Assignment 1: Object Allocator

Jump to...

Introduction to Data Structures ►