Efficiency of a Singly Linked List Composed of Bool Values Masked Into Integer Values

Joseph Garro (jmg289@uakron.edu)

Abstract:

Multiple bool values can be consolidated into a multi bit integer where each bool represents 1 bit of the integer's binary representation. Bit manipulation can be used to modify this integer and quickly retrieve any of the bool values that it is representative of. A custom data structure based on the linked list called the boolLinkedList was created in order to store these values, providing an efficient and logical approach to storing them. The performance of the boolLinkedList is comparable to STL equivalents holding bools and bitsets with respect to a given problem size N. It was found that bit manipulation did allow the boolLinkedList to operate quickly and convert integer values into the bool values that it represented, but the boolLinkedList was never as fast as the STL equivalents due to the time complexities of the algorithms required to implement its functionality. The boolLinkedList also provides a higher space efficiency compared to an array of bools as the array requires 1 byte per bool while the boolLinkedList can store 8 bools in a byte and make use of data types consisting of multiple bytes. This makes the boolLinkedList more efficient than the array provided that the size of a single node (in bytes) is less than the number of bools that are going to be stored in the array. However, the STL bool vector has an even higher space efficiency as it stores bools as individual bits, requiring a number of bytes equal to the number of bools divided by 8. Thus, the boolLinkedList is not the fastest performing or the most space efficient solution, but it compensates for these drawbacks by providing increased functionality over the STL structures that it was tested against.

Introduction:

The purpose of this research is to determine whether or not data structures can be optimized to contain specific types of data. More specifically, the goal is to determine if it is possible to create a data structure that is more efficient at storing bool values than a bitset or a bool vector in terms of space and time complexity. The size of a bool value in C++ is a byte or 8 bits, which can be determined by executing the code "std::cout << sizeof(bool);". This is the smallest size of memory that C++ (and most modern hardware and programming languages) can access (Roberts, 2015), suggesting no smaller way to store a bool value. Since bool values can be represented as a single bit with the general convention of '1' being true and '0' being false, using a byte to represent a single bool value wastes 7 bits each time this convention is used. Another way to look at this problem is with the ratio between the size in bits required to represent a bool value and the smallest memory unit that can be used to store it- a 1:8 ratio. Thus, storing a bool value wastes an amount of space equal to 7 times the space required to store it. The idea is to eliminate the wasted space by combining multiple bools together to form an integer value which will occupy the entire byte or larger amount of memory. Using bitwise manipulation, it will be possible to interact with this integer value in a manner that provides the illusion of bitwise memory access in order to take up less space and have faster access times compared to bulk bool storage. Since the efficiency and possibility of bulk information storage using this idea is the objective of this research, these loaded integers can be placed into previously studied data structures to organize them in a logical and efficient manner by taking advantage of the properties associated with the data structure used to store them. Hence, this research will determine what benefits and tradeoffs exist with such an approach to storing bool values in that manner in addition to determining its practicality.

Discussion:
Hypothesis: Combining multiple bool values together to form the binary representation of an integer will allow for greater information density and faster access times compared to storing individual bool values. These compacted bool values can be stored in a custom data structure complete with supporting operations to aid in their organization and see if better performance can be obtained compared to STL data structures that hold bool values such as the bitset (std::bitset) and bool vector (std::vector<bool>).

Binary is the basis for everything digital. Collections of '1's and '0's are arranged in meaningful patterns that allow them to represent data and for that data to be interpreted by computers. The conversion from human readable data such as numbers and characters into binary goes both directions, as humans and computers require different formats in order to understand the same information. For example, a human will recognize the number 7, but a computer has no idea what it is. Likewise, a human will not instantly recognize the number 7 seven represented in binary (0111), but a computer does. Variables in C++ are no exception to this, with integer and char values that are understandable by humans requiring conversion into binary that the computer can understand. This conversion to binary is an important part of this research, as bool values are binary, commonly denoted with a '1' for true and a '0' for false. This convention is arbitrary, but it was followed for the research documented in this paper. Additionally, all binary is this research is in unsigned representation, which restricts the decimal representation of the binary to values between 0 and (2^n) -1 (Where n is the number of bits composing the variable) for simplicity. Variables in C++ have sizes that correspond to the number of binary bits that compose them. As discussed above, a bool value is 1 byte (8 bits) while an integer is 4 bytes (32 bits). Modifiers to these variable types exist such as unsigned, which restricts the numerical range that the binary can represent to only positive values, short, which reduces the size of the variable, and long, which extends it. However, the smallest size of data that C++ can interact with is still a byte (8 bits)- the size of bool and char variables.

Despite having the same physical size in bits, the amount of information that can be represented by the bool and char variables varies significantly. The bool variable can represent either 1 (00000001) or 0 (00000000), while the char can represent all 256 possible combinations of the 8 bits, or all combinations between 11111111 and 00000000. For binary, a base 2 system is used where each digit represents a power of 2 relative to its position in the number. The right most position is 2^0 and the left most is 2^(n-1), where n is the number of digits in the binary number. The figure below shows how the powers of 2 correspond to the position of the digit in an 8 bit binary number.

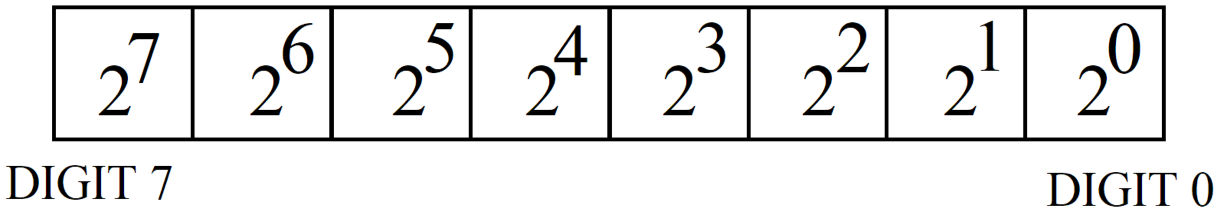


Figure 1: Powers of 2 corresponding to an 8 bit binary number.

To determine the decimal equivalent of a binary number, the powers of 2 corresponding to a ‘1’ in the binary number are summed and the powers of 2 corresponding to 0 are ignored. For example, to convert 01111110 to decimal, all powers from 2^6 to 2^1 are added:

$$2^6+2^5+2^4+2^3+2^2+2^1 = 64 + 32 + 16 + 8 + 4 + 2 = 126$$

Thus, an 8 digit binary number has decimal equivalents between 0 and 255 which correspond to the 256 unique combinations of binary digits that result in them.

Notice that the same binary ‘1’s and ‘0’s compose the bool and char variables- only the position of the bit matters. A bool can only be 1 or 0, while any combination of bits in the char can be changed. Now, suppose that each bit in the char variable 01111110 can be interpreted as a bool value. To do this, each bit composing the char is mapped to a single bool value. This turns the 8 bit char into an array storing 8 integers whose possible values are 1 or 0- the possible bool values. This idea is illustrated below in figure 2, which shows how the char variable 01111110 can be interpreted as 8 bool values.

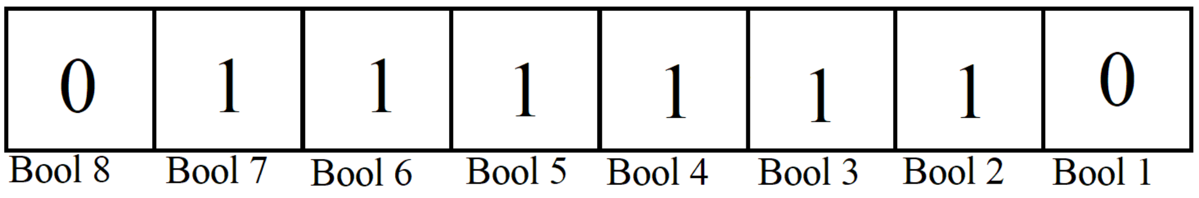


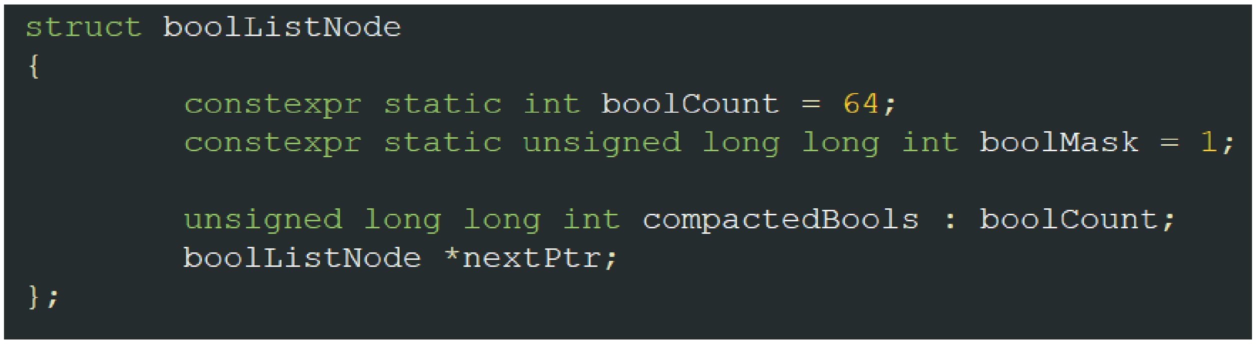
Figure 2: 8 bit binary number depicted as representing 8 bool values.

With this scheme, the 8 bit char variable can store 8 separate bool values each with their own state- 1 or 0. Since the binary composing the char determines the values in this array, changing the value of the char changes its binary representation and the collection of bool values that it represents. As discussed previously, the 8 bit char has 256 unique possible combination of bits represented by decimal values from 0 to 255. Since each bit is now equivalent to a bool value, the 8 bit char has 256 unique combinations of 8 bools which remain representable by decimal values from 0 to 255. Following the example above, the decimal number 126 could be stored as in integer to represent the unique combination bool values shown in figure 2. This idea can be extended to any type of variable, whose size in bytes determines the number of bool values that can be stored within it. Thus, the binary composing any variable, its decimal equivalent, and the bool values it represents are all equivalent. Due to this equivalence, unsigned long long integers (unsigned long long int) were used in this research to store the collection of bool values and change their values by manipulating their decimal equivalent.

The extension of using an integer to store multiple bool values is to store the integers in a data structure for logical, fast, and efficient access and storage. By storing these integers in a custom data structure, it is possible to write supporting member functions that use bitwise manipulation to provide the illusion of individual bit access- a feature otherwise unobtainable in C++. As a result, the custom data structure will allow for greater information density as whatever variable it stores can be used to store a number of bool values equal to the variables size in bits. Since bitwise manipulation operations can be performed quickly by computers, it suggests there is little overhead in manipulating integers to return integers representative of their individual bits. Conversely, as the size of the data type used to store the bools grows, the operations to return and modify individual bits will become more costly as the size of the integer increases, suggesting that this solution is only optimal up to a certain integer size. Finally, the type of data structure used to store these integers has a large effect on the time taken to access, insert, and delete the nodes containing the integers, as well as the supporting operations to manipulate the integers on a bit level. Thus, the goal of this research is to determine if this approach to storing bool variables in a custom data structure is feasible compared to using STL equivalent that hold bool values, and if so, for what sizes of integers in bits and sizes of problems is it an optimal solution.

The data structure written for this research was a singly linked list which was chosen for its ability to insert and delete nodes from anywhere within the structure. The stack forces insertions and deletions to occur at the top of the data structure, which means that only the topmost element can be removed. Therefore, any unneeded or old data underneath the top element is unable to be deleted unless it becomes the new top, requiring all elements above it to be deleted as well. This behavior is unacceptable as elements cannot be easily removed once unneeded, resulting in a data structure that can contain large amounts of old data with no way to remove it unless potentially needed data at the top is deleted as well. A queue could have been implemented as well, but the property of insertions occurring at one end and deletions occurring at the other makes deletions expensive as each element must be shifted towards the deleted element. As a result, data in the middle of the structure has a large penalty to delete as each deletion requires the shift of all of the remaining elements. Thus, the linked list provides greater flexibility in inserting and deleting its member nodes at the expense of taking slightly more time due to having no dedicated pointers or member functions to the specific locations where these actions occur.

The linked list class created for this research is called boolLinkedList. The code for the boolLinkedList class was modeled off the linked list lecture slides (U2-M3 Linked List) presented in Computer Science II, which served as a reference for how to create the structure and its basic supporting member functions. At the core of the boolLinkedList class is a private struct called boolListNode. The contents of boolListNode is shown below in Figure 3.



```
struct boolListNode
{
        constexpr static int boolCount = 64;
        constexpr static unsigned long long int boolMask = 1;

        unsigned long long int compactedBools : boolCount;
        boolListNode *nextPtr;
};
```

Figure 3: Contents of boolListNode in boolLinkedList class.

In boolListNode, there is an integer which stores the integer representation of the bool values, a pointer to the next node of the same type, and two constant static integer values used to control the attributes of each boolListNode comprising the boolLinkedList. The variable to store the integer representation of the bool values is compactedBools of type long long int. The long long in type was used as it is the largest variable available in C++ with a size of 8 bytes (64 bits), which provides support for a maximum of 64 bools. Up to 64 bools are supported as a bit field was used to declare the size in bits of compactedBools, which sets size in bits equal to the constant static value boolCount. This forces a number of bits equal to the value of boolCount to be allocated out of the 64 bits allocated for compactedBools, allowing the user of the class to specify how many bools can be stored in the node depending on their needs. As boolCount is a constant static value, it is available to all the boolListNodes that compose a boolLinkedList to ensure that each node stores the same amount of data. This value is frequently used as an argument in the class' supporting member functions as it corresponds to the number of bools in the node, allowing the internal behavior of the member functions to be automatically adjusted and scaled to produce the proper output. The second constant static value is a constant '1' that is manipulated in the class' member functions to create masks responsible for copying individual bits and performing other bitwise operations. This constant is stored as an unsigned long long integer so that it will not overflow when left shifted to create masks as it needs to be able to support up to 64 bits

The boolLinkedList class has two additional private members which are a boolListNode pointer called headPtr and an unsigned long long int called nodeCount. The pointer is used to indicate the head of the linked list and nodeCount is used to track the number of nodes currently in the boolInkedList to simplify member functions and facilitate individual node access. Finally, the boolLinkedList class has 19 public member functions that compose its interface: 2 constructors, a destructor, and 16 miscellaneous functions. These functions are summarized in Appendix A.

When using the boolLinkedList, the structure is numbered beginning from 1, which represents the boolListNode pointed to by the headPtr. Within each boolListNode, the integer value stored in compactedBools is interpreted as traditional binary, but the individual bool values are read from left to right, so the 1st bool corresponds to 2^(N-1) and the Nth bool corresponds to 2^0. This idea is shown below in Figure 4.

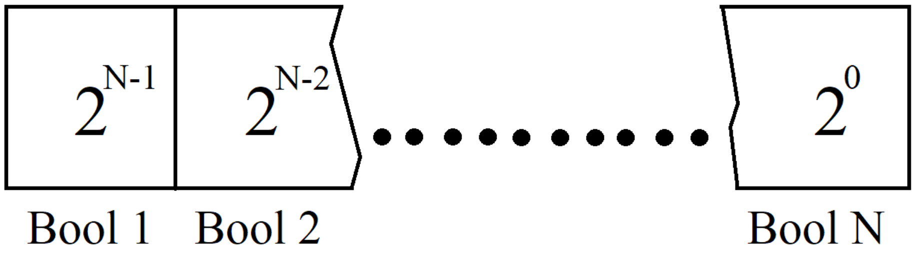


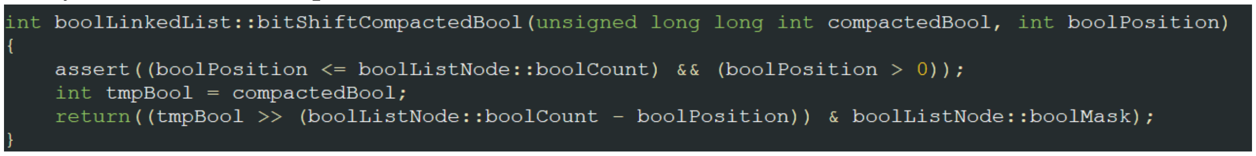
Figure 4: Depiction of how an N bit number's binary in a binaryListNode is mapped to bools.

With these properties in mind, the boolLinkedList can be treated like any other linked list consisting of standard data types.

Code Analysis:

Using the boolLinkedList class, an empty boolLinkedList is created with the default constructor which sets the headPtr to nullptr and nodeCount to 0. Anytime a node is added or deleted, nodeCount is incremented or decremented respectively to maintain a running total of all the nodes within the data structure. Concerning member functions, the bitShiftCompactedBool, isolateBool, setBoolAtPos, modifyBoolAtPos, recursiveBitShift, displayNodeBinary, displayListBinary, and flipBits functions are responsible for providing the illusion of bit manipulation in the boolLinkedList structure. The performance of these 8 functions are critical as they implement the bit level operations that will hypothetically allow the boolLinkedList structure to perform better than STL equivalents that store pure bool values (in terms of execution time and not space efficiency). Due to this importance the following section explains the time complexity of each function relative to a given problem size N.

Analysis of bitShiftCompactedBool:

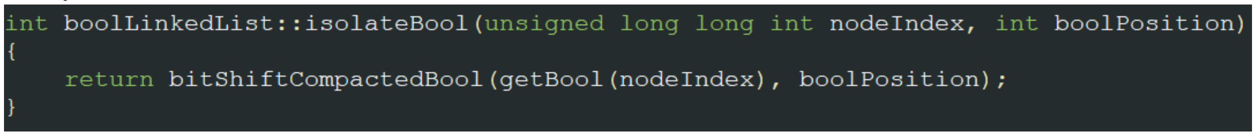


```
int boolLinkedList::bitShiftCompactedBool(unsigned long long int compactedBool, int boolPosition)
{
    assert((boolPosition <= boolListNode::boolCount) && (boolPosition > 0));
    int tmpBool = compactedBool;
    return((tmpBool >> (boolListNode::boolCount - boolPosition)) & boolListNode::boolMask);
}
```

Figure 5: Code for bitShiftCompactedBool function.

The bitShiftCompactedBool function takes an unsigned long long integer stored in a boolListNode and an integer that corresponds to the desired position in the node. The compactedBool value is backed up and the backup is then shifted right so that the desired position can align with the boolMask and the result is returned. The worst case for this function is a shift (N-1) times to get the 1st bool, but this can be done in one step => O(1).

Analysis of isolateBool:

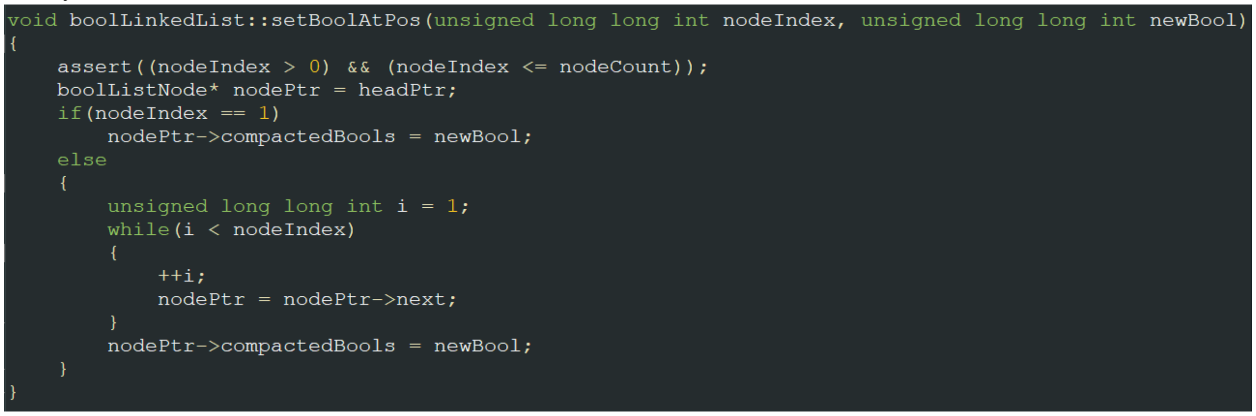


```
int boolLinkedList::isolateBool(unsigned long long int nodeIndex, int boolPosition)
{
    return bitShiftCompactedBool(getBool(nodeIndex), boolPosition);
}
```

Figure 6: Code for isolateBool function.

The isolateBool functions takes an unsigned long long int node index and the desired position of the bool stored in that node as arguments. This function invokes the bitShiftCompactedBool function using the getBool function with the node indexed passed to it, as well as the desired bool’s position. Since the getBool function is used, the boolLinkedList must be traversed to the desired node from the beginning. The worst case scenario is requesting the last node’s compactedBool, which requires N iterations to get to. As this action must be completed to invoke the bitShiftCompactedBool function call, the isolateBool function’s runtime depends on the size of the problem => O(N).

Analysis of setBoolAtPos:

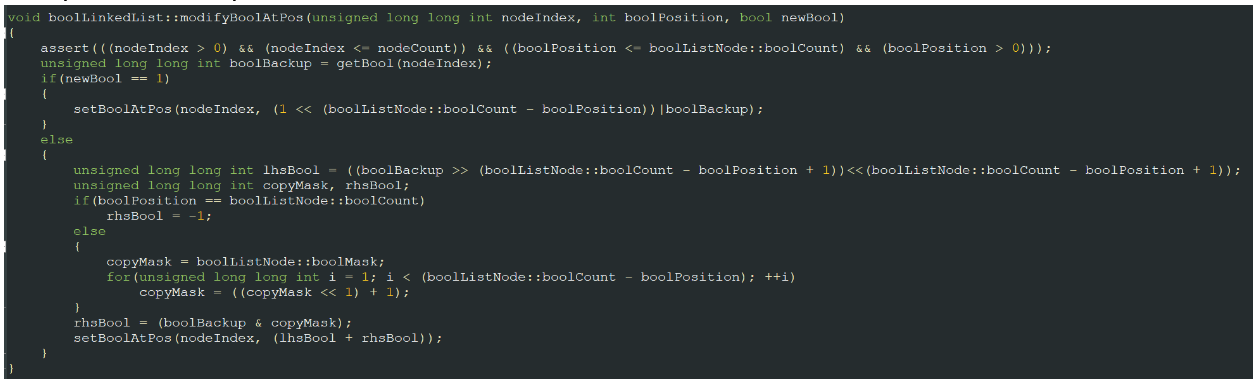


```
void boolLinkedList::setBoolAtPos(unsigned long long int nodeIndex, unsigned long long int newBool)
{
    assert((nodeIndex > 0) && (nodeIndex <= nodeCount));
    boolListNode* nodePtr = headPtr;
    if(nodeIndex == 1)
        nodePtr->compactedBools = newBool;
    else
    {
        unsigned long long int i = 1;
        while(i < nodeIndex)
        {
            ++i;
            nodePtr = nodePtr->next;
        }
        nodePtr->compactedBools = newBool;
    }
}
```

Figure 7: Code for setBoolAtPos function.

The setBoolAtPos function takes 2 unsigned long long int values: a node index and a new value that will replace the value stored at the supplied index. In the worst case scenario, the last node will be replaced which requires traveling to the end of the boolLinkedList, which takes N iterations. Thus, the runtime of this functions depends on the size of the problem => O(N).

Analysis of modifyBoolAtPos:

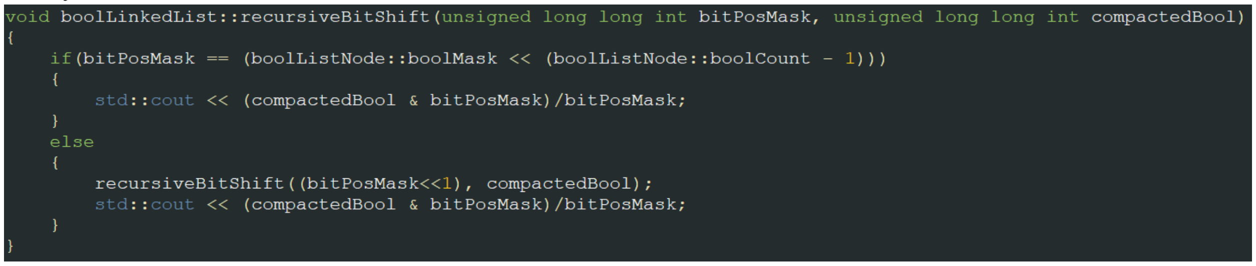


```
void boolLinkedList::modifyBoolAtPos(unsigned long long int nodeIndex, int boolPosition, bool newBool)
{
    assert(((nodeIndex > 0) && (nodeIndex <= nodeCount)) && ((boolPosition <= boolListNode::boolCount) && (boolPosition > 0)));
    unsigned long long int boolBackup = getBool(nodeIndex);
    if(newBool == 1)
    {
        setBoolAtPos(nodeIndex, (1 << (boolListNode::boolCount - boolPosition))|boolBackup);
    }
    else
    {
        unsigned long long int lhsBool = ((boolBackup >> (boolListNode::boolCount - boolPosition + 1))<<(boolListNode::boolCount - boolPosition + 1));
        unsigned long long int copyMask, rhsBool;
        if(boolPosition == boolListNode::boolCount)
            rhsBool = -1;
        else
        {
            copyMask = boolListNode::boolMask;
            for(unsigned long long int i = 1; i < (boolListNode::boolCount - boolPosition); ++i)
                copyMask = ((copyMask << 1) + 1);
        }
        rhsBool = (boolBackup & copyMask);
        setBoolAtPos(nodeIndex, (lhsBool + rhsBool));
    }
}
```

Figure 8: Code for modifyBoolAtPos function.

The setBoolAtPos function takes the index of a boolListNode, the position of a bool within the node, and the new value of the bool as arguments to allow the user to change the value of any bool in any node. For this to work, masks are created to either side of the bool that is changed, the bool is changed, and then the masks are anded together to form the result. Since the setBoolAtPos function is used to assign the resulting compactedBool, the modifyBoolAtPos also has a linear time complexity assuming that the result is assigned to the final node => O(N).

Analysis of recursiveBitShift:

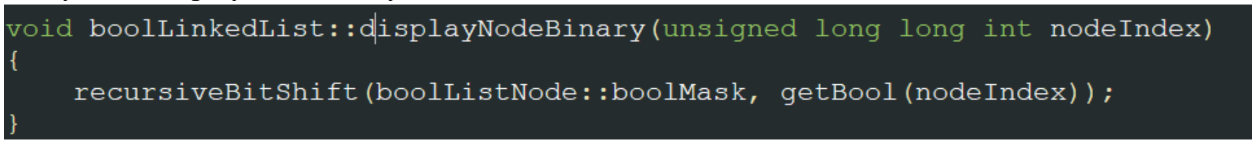


```
void boolLinkedList::recursiveBitShift(unsigned long long int bitPosMask, unsigned long long int compactedBool)
{
    if(bitPosMask == (boolListNode::boolMask << (boolListNode::boolCount - 1)))
    {
        std::cout << (compactedBool & bitPosMask)/bitPosMask;
    }
    else
    {
        recursiveBitShift((bitPosMask<<1), compactedBool);
        std::cout << (compactedBool & bitPosMask)/bitPosMask;
    }
}
```

Figure 9: Code for recursiveBitShift function.

The code for recursiveBitShift accepts 2 unsigned long long ints representing a compactedBool and a mask used to isolate individual bits of the compactedBool. This function is called recursively until the mask is equal to the number of bits composing the compactedBool, at which point the function is unwound to display the binary within the node. This function is independent of the problem size N, but is dependent on the number of bits that compose N, so the time complexity is still dependent on something => O(m), where m is the number of bits in the compactedBool (equal to boolListNode::boolcount in the boolLinkedList.h file).

Analysis of displayNodeBinary:

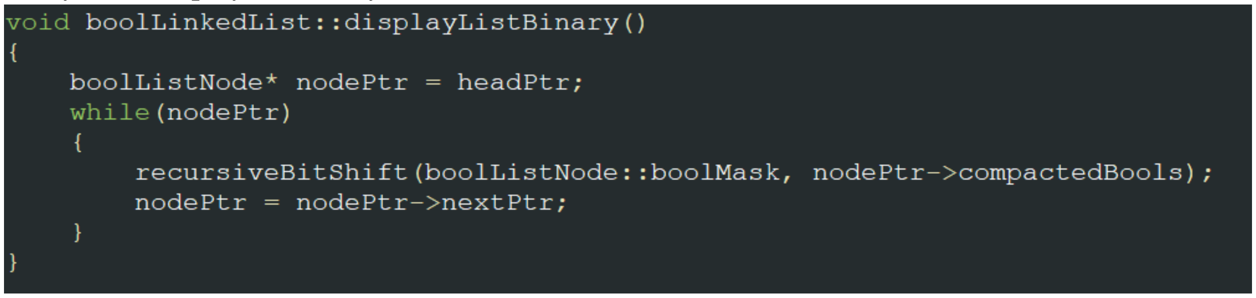


```
void boolLinkedList::displayNodeBinary(unsigned long long int nodeIndex)
{
    recursiveBitShift(boolListNode::boolMask, getBool(nodeIndex));
}
```

Figure 10: Code for displayNodeBinary function.

The displayNodeBinary function accepts a node index as a parameter and passes it to the recursiveBitShift function along with the mask stored in the boolLinkedList class. Since displayNodeBinary invokes the recursiveBitShift function with a getBool parameter using the passed nodeIndex, the worst case scenario is that the last node in the boolLinkedList is accessed, which will require traversing the entire data structure=> O(N).

Analysis of displayListBinary:

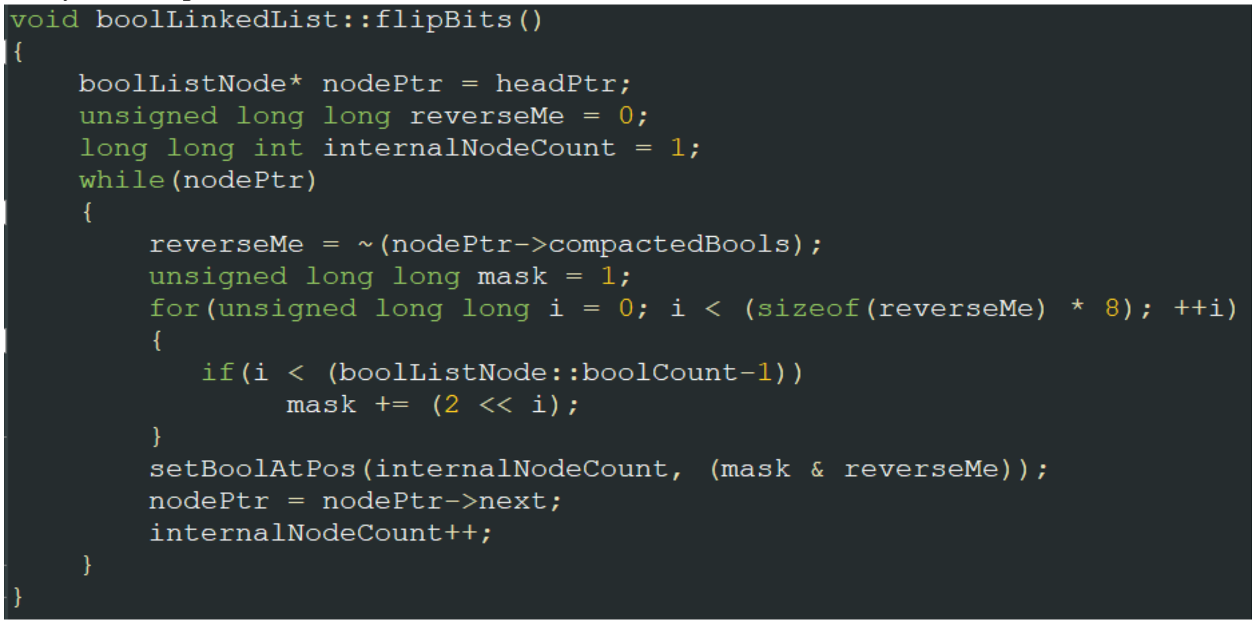


```
void boolLinkedList::displayListBinary()
{
    boolListNode* nodePtr = headPtr;
    while(nodePtr)
    {
        recursiveBitShift(boolListNode::boolMask, nodePtr->compactedBools);
        nodePtr = nodePtr->nextPtr;
    }
}
```

Figure 11: Code for dislpayListBinary function.

The displayListBinary function interates from the headPtr through the end of the boolLinkedList, calling the recursiveBitShiftFunction with the mask stored in the boolLinkedList class and the bool stored in the current node. Every element in the boolLinkedList is accessed when running this program, so the time complexity is dependent on the size of the problem N => O(N).

Analysis of flipBits:



```
void boolLinkedList::flipBits()
{
    boolListNode* nodePtr = headPtr;
    unsigned long long reverseMe = 0;
    long long int internalNodeCount = 1;
    while(nodePtr)
    {
        reverseMe = ~(nodePtr->compactedBools);
        unsigned long long mask = 1;
        for(unsigned long long i = 0; i < (sizeof(reverseMe) * 8); ++i)
        {
           if(i < (boolListNode::boolCount-1))
                mask += (2 << i);
        }
        setBoolAtPos(internalNodeCount, (mask & reverseMe));
        nodePtr = nodePtr->next;
        internalNodeCount++;
    }
}
```

Figure 12: Code for flipBits function.

The flipBits function iterates through the entire boolLinkedList and inverts the value stored in boolListNode's compactedBools value to invert the values of the bools it represents. This function starts at the headPtr and works its way through each node of the boolLinkedList. To maintain compatibility with bit sizes less than 64 bits, a mask is used to isolate the desired portion of the inverted compactedBools variable. This mask is anded with the inverted compactedBools variable and then assigned to the corresponding node using an internally incremented node counter and the setBoolAtPos function. There is no worst case scenario for this function as the entire boolLinkedList is traversed so it can be inverted and the entire list must be traversed N times in order to set the new compactedBoolsValue. Thus, each call to flipBits requires two dependent traversals of the entire linked list up to the current node => O(N^2).

Testing Methodology:
To test the performance of the boolLinkedList data structure, the example test case code provided on Brightspace called "Complexity_Runtime_DS_Example" was modified to create new test code. Following the format of the original code, the modified code performs a series of tests over three different data structures: an STL list composed of bitsets, an STL vector storing bools, and the boolLinkedList. The bitset list is a close approximation to the boolLinkedList as bitsets store a sequence of bits of set length that can be accessed and manipulated, and using the STL list to store them provides similar functionality to the boolLinkedList. The STL vector containing bools, or the bool vector, is another close approximation but can only store one bool value per element while the boolLinkedList and list of bitsets can store multiple. Therefore, the question is whether or not the boolLinkedList can provide a faster means of operation than the list of bitsets and bool vector due to using bit operations, as well as if it can provide a more space efficient solution than either structure.All 3 data structures were tested by recording the time it took them to complete a specified task using the provided "Complexity_Recorder.hpp" and

"Complexity_Timer.hpp" files. To use the timer, each test had a set number of trials and was repeated a certain number of times for a given problem size. For the testing, the number of trials was set to 5 and the number of repetitions was set to 4. The recorder records the total time it takes to complete all repetition for each trial, selects the median time of the trials, and then divides that time by the number of repetitions to use as the result. The problem size N was defined in terms of a lower bound and an upper bound which were equal to 1 and 2048 respectively. The problem size was calculated by taking the lower bound and multiplying it by a constant factor equal to 20. After the series of 4 tests was performed on each structure, the lower limit of the problem size was doubled and again multiplied by 20 to create the new problem size on which another 4 repetitions of the test were ran. This cycle would repeat until the lower bound was greater than the upper bound. The variables for the upper and lower bounds of the sequence, the number of trials, the number of structures (used in order to individually test structures with a for loop), the factor, and the number of repetitions to perform are all stored as constants at the top of the program so that they can be easily changed in one location to produce changes that are reflected in the entire testing program.

Due to the size different between the bool vector and other 2 structures, the bool vector had to be tested with K*N elements in order to have a uniform problem size across all structures, where K is the number of bools that the boolLinkedList and list of bitsets can store in a single element. Hence, the N * number of bools in a node yields the total number of bools represented in the boolLinkedList and bitset list. To control the size of the bitset list's bitsets, a constant called BOOLSIZE was added to the test program whose value is manually declared to match the boolCount value found in the boolLinkedList.h file. Prior to any test being recorded, each structure was loaded with the appropriate amount of test data that was randomly generated. To randomly generates bools for the bool vector, (rand() % 2) was used to generate a random 1 or 0. To generate random integer values corresponding to collection of a specific number of bools, (rand() % TWOPOWERSUBONE + 1) was used where TWOPOWERSUBONE is a constant corresponding to (2^N)-1 (N is the number of bits that the structures will hold). This yields an integer in the range of 0 to 2^(N-1) which is storable in either structure to represent N bool values. The problem here is that rand() generates a random value up to RAND_MAX, which "is at least 32767" (RAND_MAX), much smaller than the 16, 32, and 64 bit compactedBools. At this point, random values are either not generated or their range is super limited, but it still serves as a way to populate the boolLinkedList and STL list with acceptable values.

For testing, each data structure was subjected to the same 4 tests:

1. The time to print all bool values in the data structure
2. The time to access and print the last bool value stored in the data structure
3. The time to flip all bool values within the structure
4. The time to completely delete the structure when filled with data

The reasoning behind these tests was that they can be used to determine how the time to output all the data in the data structure, the time to access and print the value stored in the last node, the time to flip all the values within the data structure, and the time to completely delete the data structure are influenced by the problem size N. As these tests access either all elements or solely the last element, they should represent the worst case scenario in terms of performing an action to the entire structure (opposed to only part of it) or only on the last node, which should be the most time consuming to access. Therefore, typical performance of any of the structures can be expected to be better than the times obtained as a result of this research.

The first test iterates through the STL list and vector using iterators and displays their contents to the console. Likewise, the boolLinkedList's displayListBinary member function is used to iterate through all nodes and output their content. For the second test, the STL list's back member function is used to access its last element, which is compared to the STl vector's element access using "[]' and the boolLinkedList's displayNodeBinary member function, which must iterate through all nodes to reach the end. The third test is based on the ability to flip the collection of bits stored in the bool vector or in a bitset. This test is based on the flip member function implemented in the bitset and bool vector. However, since the STL list is storing bitsets, iterators must be used to traverse the list and the flip operation must be performed on each node (or N times), while only one flip operation needs to be performed on the bool vector. Meanwhile, the boolLinkedList's flipBits function is used to traverse through each node and flip their bits (which required N iterations as well). Finally, the 4th test measures the time to delete each structure when full using their respective commands to delete individual elements (not to clear structures). For the delete test, the STL vector's pop_back function was compared to the STL list's pop_front function, while the boolLinkedList repeatedly deleted its front element until empty. This method of testing the deletion of each data structure was decided as writing code to make the vector repeatedly delete its front element and boolLinkedList delete its rear element resulted in unacceptable execution times. Furthermore, repeatedly deleting the front element of the boolLinkedList is an optimal way to delete it, so its length should solely depend on the problem size during its test.

Testing was performed via a switch statement that allows the user to select the desired test to perform. Due to this, data structure creation and population with random values is local to each switch case which results in long execution times of the main.cpp test code but ensures that the structures are deleted when they leave scope, so no test will interfere with the results of another.

Results:
The research has two results. The first result is the run time recorded for the tests performed on each data structure, which can be used to determine what data structure that had the best performance when executing each task. The second result is of the space complexity of each structure, which determines the amount of space needed by the structure and the point at which the structure is more efficient than an array containing the same type of data.

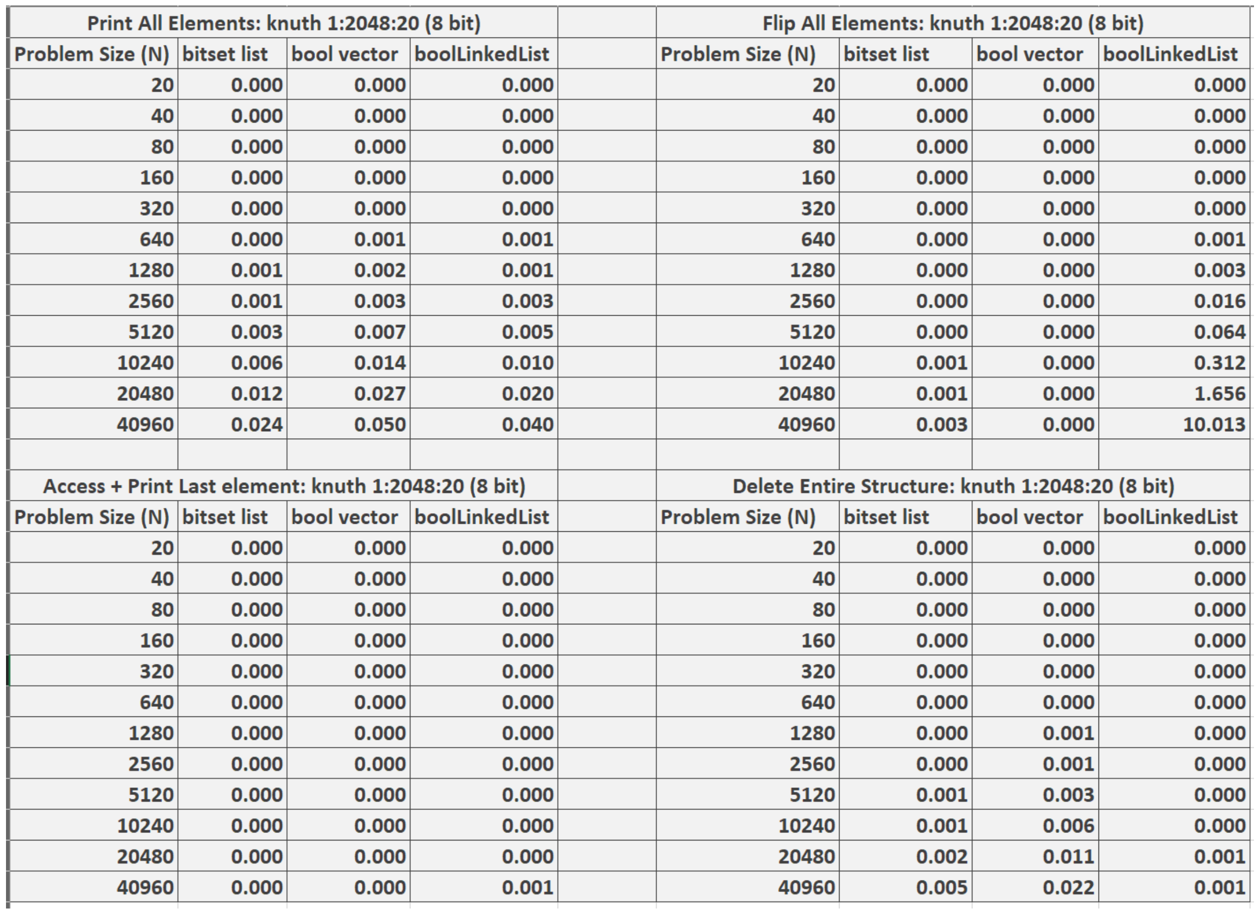
For the run time results, the program was executed on the CS Department Server for values of boolCount in the boolLinkedList.h file and BOOLCOUNT in the main.cpp file equal to 8, 16, 32 and 64. As stated above, the factor was set to 20, the number of repetitions 4, the lower bound of the sequence 1, and the higher bound of the sequence 2048. Finally, the constant TWOPOWERSUBONE was set according to the value of BOOLCOUNT. Combinations of BOOLCOUNT and TWOPOWERSUBONE are shown below in Figure 13, as well as the required value of boolCount in the boolLinkedList.h file.

boolLinkedList.h	main.cpp

boolCount	BOOLCOUNT	TWOPOWERSUBONE
8	8	255
16	16	65535
32	32	4294967295
64	64	18446744073709551615

Figure 13: Table depicting the combinations of boolCount, BOOLCOUNT, and TWOPOWERSUBONE that were tested.

The runtime results of the trials are shown in figures 14 through 17 below:



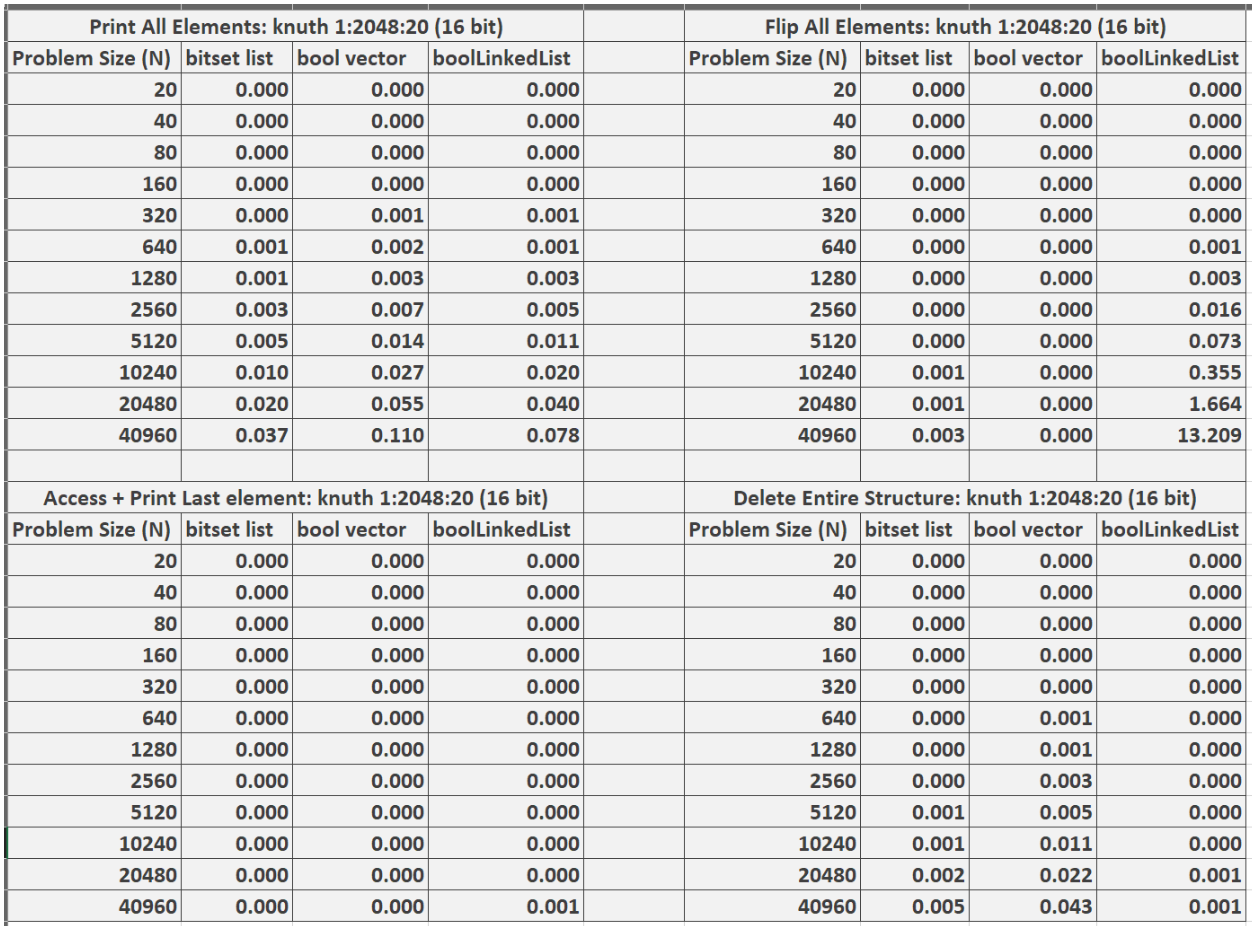
Print All Elements: knuth 1:2048:20 (8 bit)			
Problem Size (N)	bitset list	bool vector	boolLinkedList
20	0.000	0.000	0.000
40	0.000	0.000	0.000
80	0.000	0.000	0.000
160	0.000	0.000	0.000
320	0.000	0.000	0.000
640	0.000	0.001	0.001
1280	0.001	0.002	0.001
2560	0.001	0.003	0.003
5120	0.003	0.007	0.005
10240	0.006	0.014	0.010
20480	0.012	0.027	0.020
40960	0.024	0.050	0.040

Flip All Elements: knuth 1:2048:20 (8 bit)			
Problem Size (N)	bitset list	bool vector	boolLinkedList
20	0.000	0.000	0.000
40	0.000	0.000	0.000
80	0.000	0.000	0.000
160	0.000	0.000	0.000
320	0.000	0.000	0.000
640	0.000	0.000	0.001
1280	0.000	0.000	0.003
2560	0.000	0.000	0.016
5120	0.000	0.000	0.064
10240	0.001	0.000	0.312
20480	0.001	0.000	1.656
40960	0.003	0.000	10.013

Access + Print Last element: knuth 1:2048:20 (8 bit)			
Problem Size (N)	bitset list	bool vector	boolLinkedList
20	0.000	0.000	0.000
40	0.000	0.000	0.000
80	0.000	0.000	0.000
160	0.000	0.000	0.000
320	0.000	0.000	0.000
640	0.000	0.000	0.000
1280	0.000	0.000	0.000
2560	0.000	0.000	0.000
5120	0.000	0.000	0.000
10240	0.000	0.000	0.000
20480	0.000	0.000	0.000
40960	0.000	0.000	0.001

Delete Entire Structure: knuth 1:2048:20 (8 bit)			
Problem Size (N)	bitset list	bool vector	boolLinkedList
20	0.000	0.000	0.000
40	0.000	0.000	0.000
80	0.000	0.000	0.000
160	0.000	0.000	0.000
320	0.000	0.000	0.000
640	0.000	0.000	0.000
1280	0.000	0.001	0.000
2560	0.000	0.001	0.000
5120	0.001	0.003	0.000
10240	0.001	0.006	0.000
20480	0.002	0.011	0.001
40960	0.005	0.022	0.001

Figure 14: Runtime results of each test on each structure when each node was configured to store 8 bits.



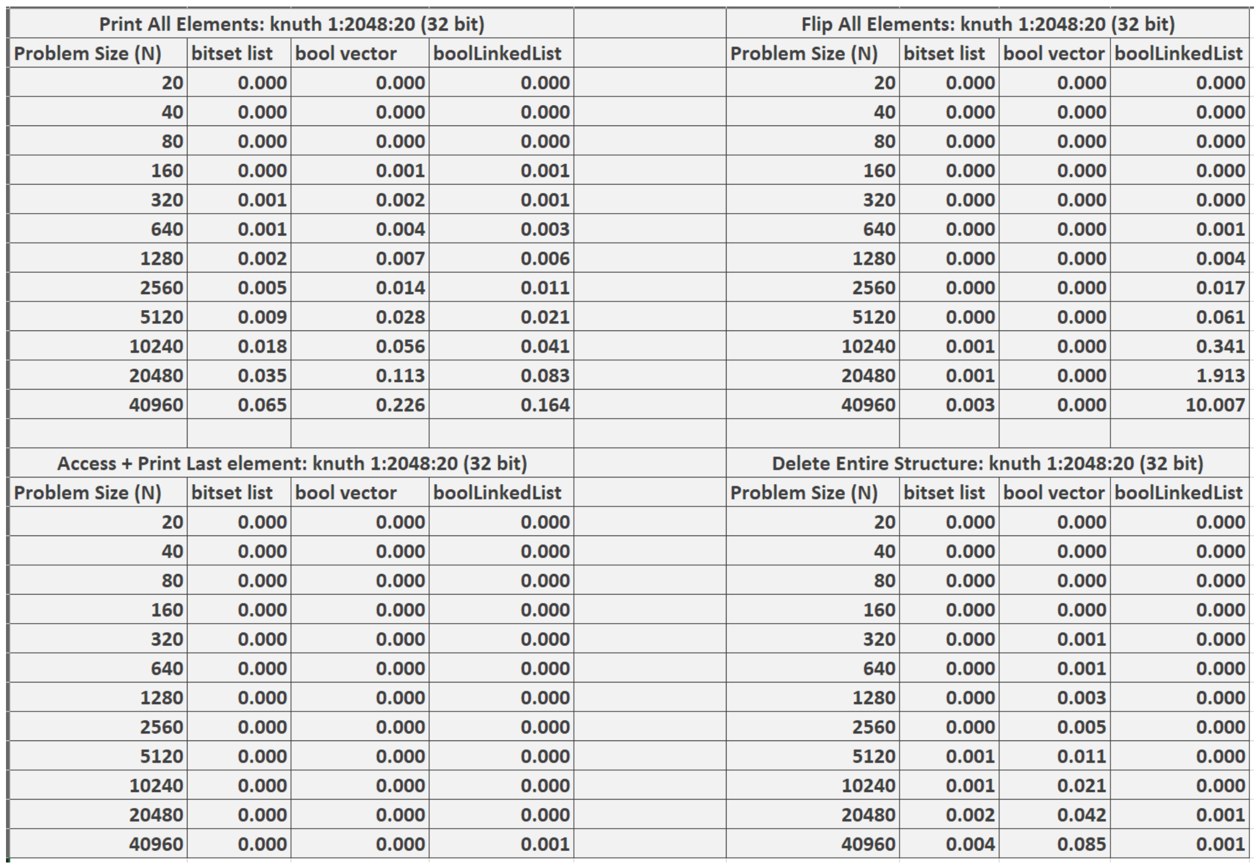
Print All Elements: knuth 1:2048:20 (16 bit)			
Problem Size (N)	bitset list	bool vector	boolLinkedList
20	0.000	0.000	0.000
40	0.000	0.000	0.000
80	0.000	0.000	0.000
160	0.000	0.000	0.000
320	0.000	0.001	0.001
640	0.001	0.002	0.001
1280	0.001	0.003	0.003
2560	0.003	0.007	0.005
5120	0.005	0.014	0.011
10240	0.010	0.027	0.020
20480	0.020	0.055	0.040
40960	0.037	0.110	0.078

Flip All Elements: knuth 1:2048:20 (16 bit)			
Problem Size (N)	bitset list	bool vector	boolLinkedList
20	0.000	0.000	0.000
40	0.000	0.000	0.000
80	0.000	0.000	0.000
160	0.000	0.000	0.000
320	0.000	0.000	0.000
640	0.000	0.000	0.001
1280	0.000	0.000	0.003
2560	0.000	0.000	0.016
5120	0.000	0.000	0.073
10240	0.001	0.000	0.355
20480	0.001	0.000	1.664
40960	0.003	0.000	13.209

Access + Print Last element: knuth 1:2048:20 (16 bit)			
Problem Size (N)	bitset list	bool vector	boolLinkedList
20	0.000	0.000	0.000
40	0.000	0.000	0.000
80	0.000	0.000	0.000
160	0.000	0.000	0.000
320	0.000	0.000	0.000
640	0.000	0.000	0.000
1280	0.000	0.000	0.000
2560	0.000	0.000	0.000
5120	0.000	0.000	0.000
10240	0.000	0.000	0.000
20480	0.000	0.000	0.000
40960	0.000	0.000	0.001

Delete Entire Structure: knuth 1:2048:20 (16 bit)			
Problem Size (N)	bitset list	bool vector	boolLinkedList
20	0.000	0.000	0.000
40	0.000	0.000	0.000
80	0.000	0.000	0.000
160	0.000	0.000	0.000
320	0.000	0.000	0.000
640	0.000	0.001	0.000
1280	0.000	0.001	0.000
2560	0.000	0.003	0.000
5120	0.001	0.005	0.000
10240	0.001	0.011	0.000
20480	0.002	0.022	0.001
40960	0.005	0.043	0.001

Figure 15: Runtime results of each test on each structure when each node was configured to store 16 bits.



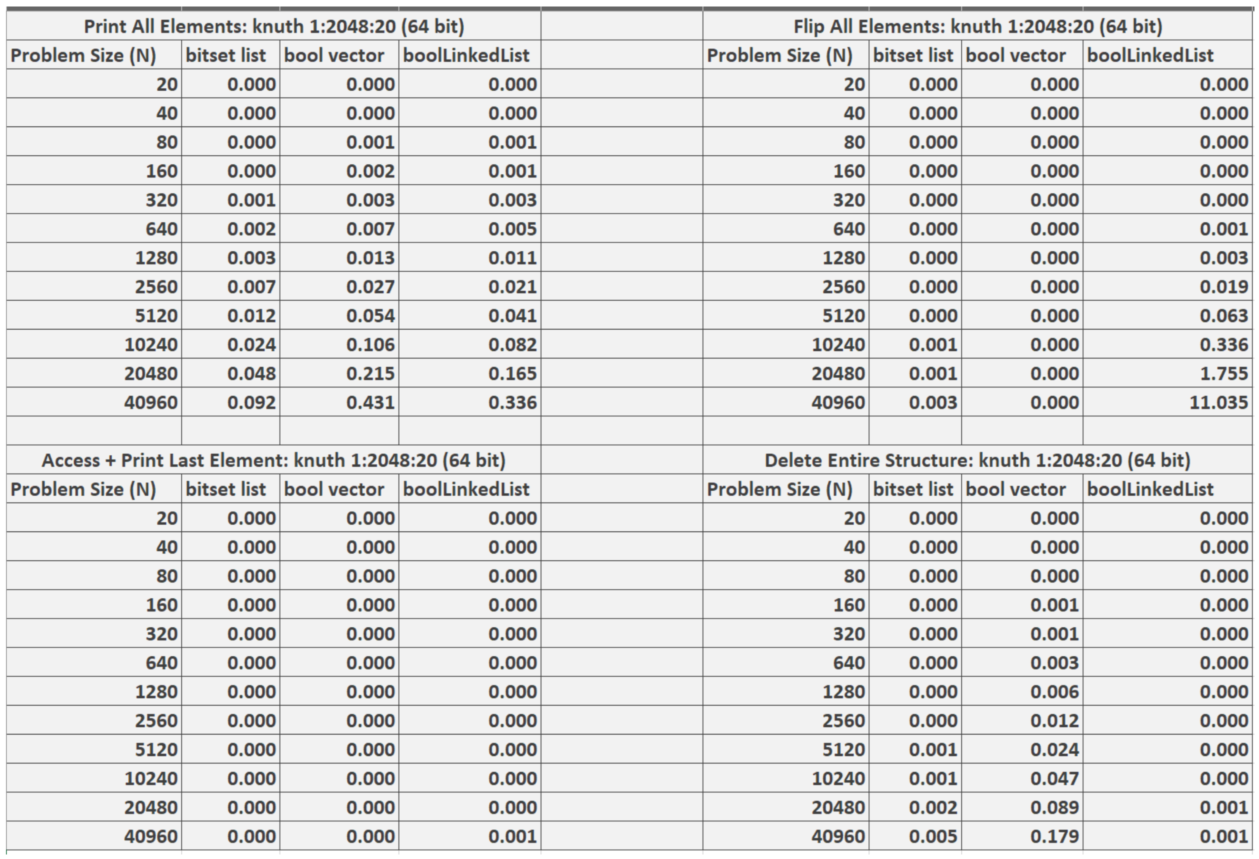
Print All Elements: knuth 1:2048:20 (32 bit)			
Problem Size (N)	bitset list	bool vector	boolLinkedList
20	0.000	0.000	0.000
40	0.000	0.000	0.000
80	0.000	0.000	0.000
160	0.000	0.001	0.001
320	0.001	0.002	0.001
640	0.001	0.004	0.003
1280	0.002	0.007	0.006
2560	0.005	0.014	0.011
5120	0.009	0.028	0.021
10240	0.018	0.056	0.041
20480	0.035	0.113	0.083
40960	0.065	0.226	0.164

Flip All Elements: knuth 1:2048:20 (32 bit)			
Problem Size (N)	bitset list	bool vector	boolLinkedList
20	0.000	0.000	0.000
40	0.000	0.000	0.000
80	0.000	0.000	0.000
160	0.000	0.000	0.000
320	0.000	0.000	0.000
640	0.000	0.000	0.001
1280	0.000	0.000	0.004
2560	0.000	0.000	0.017
5120	0.000	0.000	0.061
10240	0.001	0.000	0.341
20480	0.001	0.000	1.913
40960	0.003	0.000	10.007

Access + Print Last element: knuth 1:2048:20 (32 bit)			
Problem Size (N)	bitset list	bool vector	boolLinkedList
20	0.000	0.000	0.000
40	0.000	0.000	0.000
80	0.000	0.000	0.000
160	0.000	0.000	0.000
320	0.000	0.000	0.000
640	0.000	0.000	0.000
1280	0.000	0.000	0.000
2560	0.000	0.000	0.000
5120	0.000	0.000	0.000
10240	0.000	0.000	0.000
20480	0.000	0.000	0.000
40960	0.000	0.000	0.001

Delete Entire Structure: knuth 1:2048:20 (32 bit)			
Problem Size (N)	bitset list	bool vector	boolLinkedList
20	0.000	0.000	0.000
40	0.000	0.000	0.000
80	0.000	0.000	0.000
160	0.000	0.000	0.000
320	0.000	0.001	0.000
640	0.000	0.001	0.000
1280	0.000	0.003	0.000
2560	0.000	0.005	0.000
5120	0.001	0.011	0.000
10240	0.001	0.021	0.000
20480	0.002	0.042	0.001
40960	0.004	0.085	0.001

Figure 16: Runtime results of each test on each structure when each node was configured to store 32 bools.



Print All Elements: knuth 1:2048:20 (64 bit)			
Problem Size (N)	bitset list	bool vector	boolLinkedList
20	0.000	0.000	0.000
40	0.000	0.000	0.000
80	0.000	0.001	0.001
160	0.000	0.002	0.001
320	0.001	0.003	0.003
640	0.002	0.007	0.005
1280	0.003	0.013	0.011
2560	0.007	0.027	0.021
5120	0.012	0.054	0.041
10240	0.024	0.106	0.082
20480	0.048	0.215	0.165
40960	0.092	0.431	0.336

Flip All Elements: knuth 1:2048:20 (64 bit)			
Problem Size (N)	bitset list	bool vector	boolLinkedList
20	0.000	0.000	0.000
40	0.000	0.000	0.000
80	0.000	0.000	0.000
160	0.000	0.000	0.000
320	0.000	0.000	0.000
640	0.000	0.000	0.001
1280	0.000	0.000	0.003
2560	0.000	0.000	0.019
5120	0.000	0.000	0.063
10240	0.001	0.000	0.336
20480	0.001	0.000	1.755
40960	0.003	0.000	11.035

Access + Print Last Element: knuth 1:2048:20 (64 bit)			
Problem Size (N)	bitset list	bool vector	boolLinkedList
20	0.000	0.000	0.000
40	0.000	0.000	0.000
80	0.000	0.000	0.000
160	0.000	0.000	0.000
320	0.000	0.000	0.000
640	0.000	0.000	0.000
1280	0.000	0.000	0.000
2560	0.000	0.000	0.000
5120	0.000	0.000	0.000
10240	0.000	0.000	0.000
20480	0.000	0.000	0.000
40960	0.000	0.000	0.001

Delete Entire Structure: knuth 1:2048:20 (64 bit)			
Problem Size (N)	bitset list	bool vector	boolLinkedList
20	0.000	0.000	0.000
40	0.000	0.000	0.000
80	0.000	0.000	0.000
160	0.000	0.001	0.000
320	0.000	0.001	0.000
640	0.000	0.003	0.000
1280	0.000	0.006	0.000
2560	0.000	0.012	0.000
5120	0.001	0.024	0.000
10240	0.001	0.047	0.000
20480	0.002	0.089	0.001
40960	0.005	0.179	0.001

Figure 17: Runtime results of each test on each structure when each node was configured to store 64 bools.

As shown in Figures 11 through 14, the problem size was not large enough to produce measurable execution times for accessing and printing the last element in each structure for all data structures tested. Concerning other tests, the custom boolLinkedList had the best execution time when deleting the entire structure. The list of bitsets was best at printing all values within the data structure, while the bool vector was fastest at flipping all the bool values within the structure. Interestingly, each data structure has a test where it outperformed the others independent of problem size, suggesting that a data structure can be recommended by the work that is expected to be performed on the data it contains. For instance, when the data structure needs to be frequently deleted, the boolLinkedList will be an optimal solution after a problem size of 5120 according to each test. Likewise, an application requiring that the values of all the bool values within the data structure be inversed frequently would save large amounts of time by using the bool vector compared to the other structures. This is not to say that the other data structures will perform poorly in these cases, as their performance all around is fast and close to one another, but that one data structure appears to be optimized for each tested task. Finally, comparing the execution times for each test across each data structure as a function of the problem size and number of bools that it stored produced the plots shown below in figures 18 through 21.

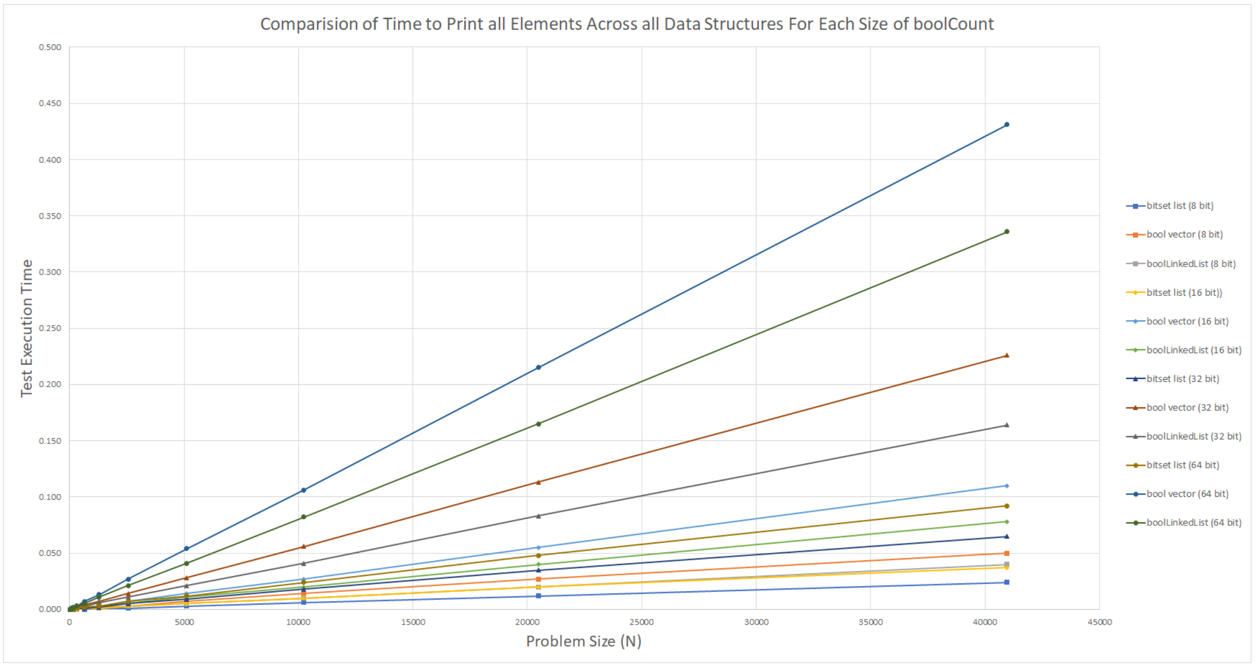


Figure 18: Chart plotting all execution times for test 1 across all 3 data structures for with respect to the problem size N for 8, 16, 32, and 64 bit bool values.

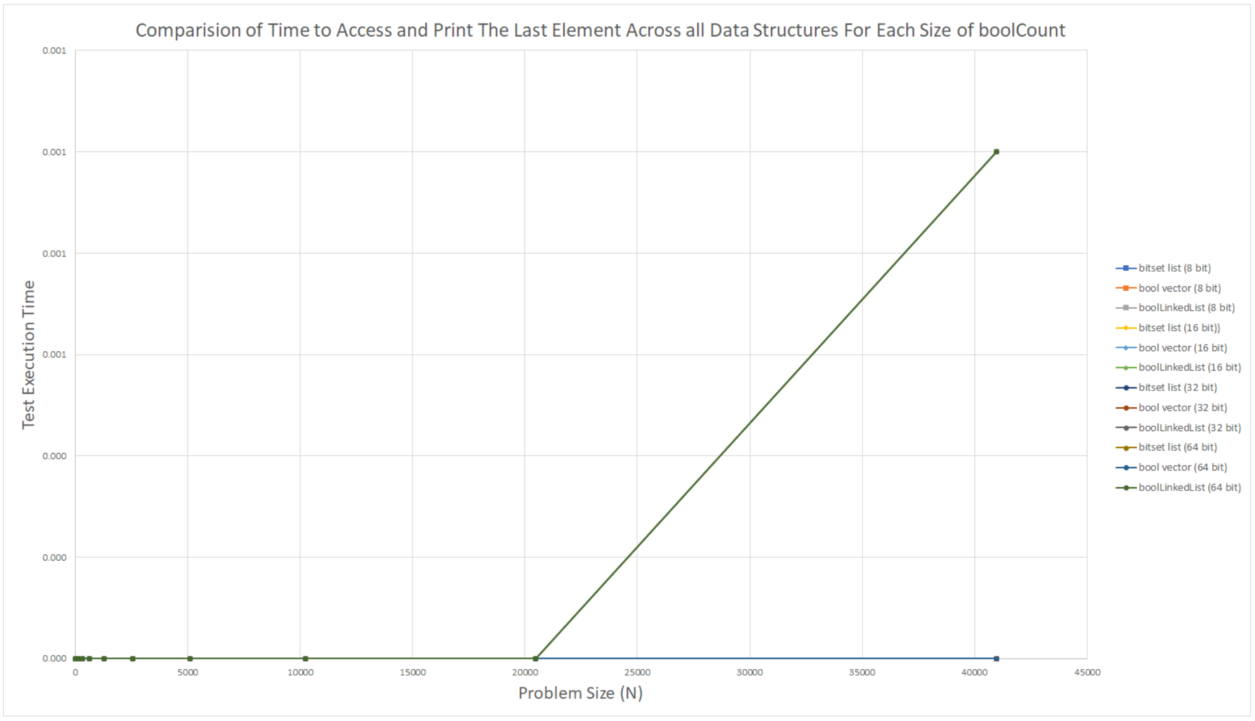


Figure 19: Chart plotting all execution times for test 2 across all 3 data structures for with respect to the problem size N for 8, 16, 32, and 64 bit bool values.

I'm

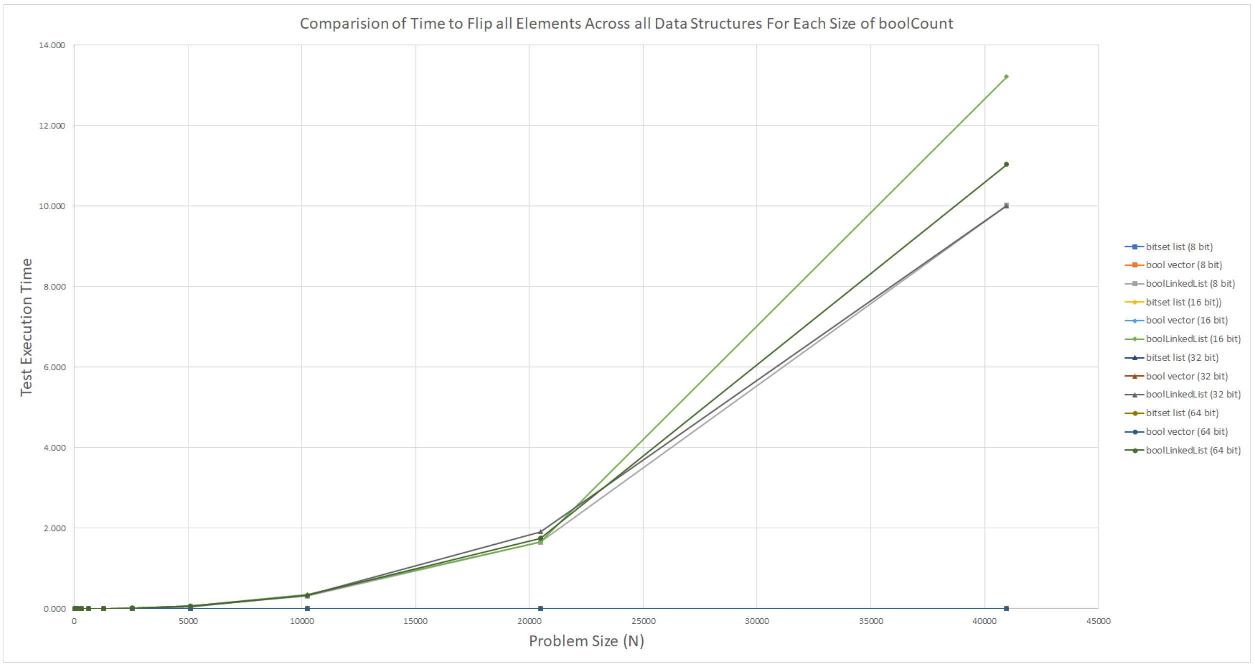


Figure 20: Chart plotting all execution times for test 3 across all 3 data structures for with respect to the problem size N for 8, 16, 32, and 64 bit bool values.

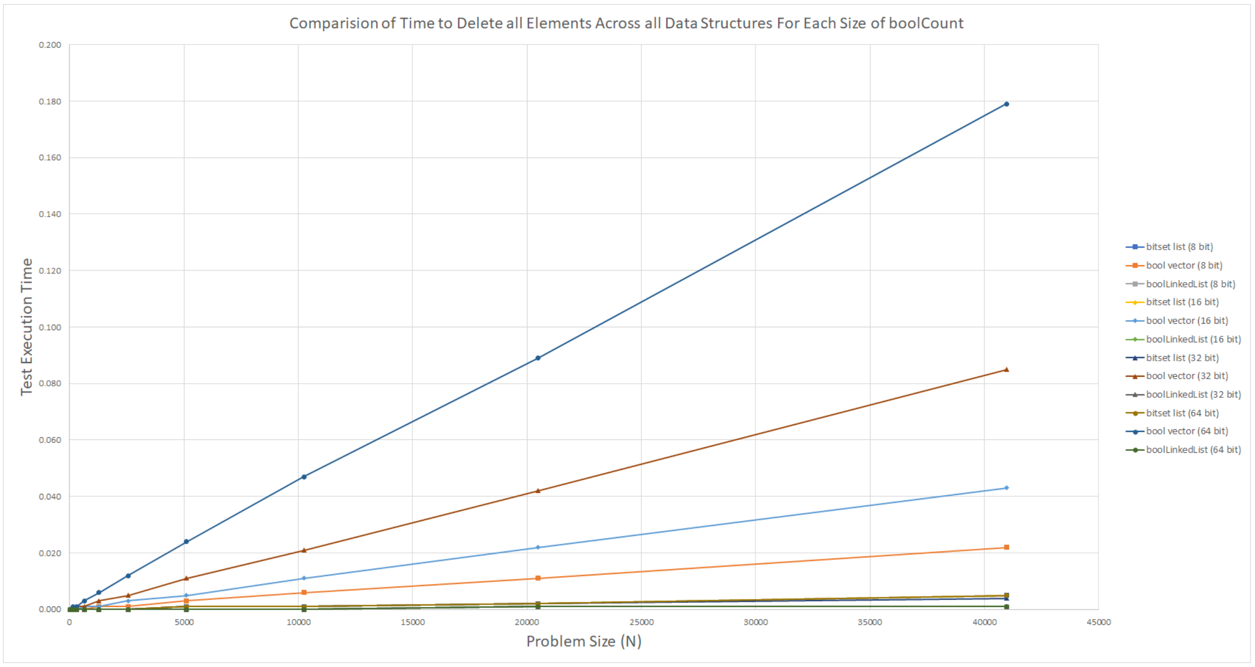


Figure 21: Chart plotting all execution times for test 4 across all 3 data structures for with respect to the problem size N for 8, 16, 32, and 64 bit bool values.

These plots allow the runtimes for each test to be viewed graphically as a function of N and demonstrate how runtime was dependent on N as well as the number of bools each data structure stored in an element. From these graphs, the complexity of each test on each structure can be observed, as well as where the functions began to deviate from each other in terms of performance with respect to the problem size. In figure 18, the run time of test 1 on each tested data structure was shown grow linearly with respect to the problem size, having a time complexity of O(N). Concerning the performance of each structure, performance was near identical up to N = 1280, at which point the 32 and 64 bit bool vectors and boolLinkedLists separated from the rest of the structures and had a much larger run time. Of the structures grouped at the bottom of the plot, the list storing bitsets had the fastest execution time for each size of the integer that was tested, making it the idea data structure to use if large amounts of bool values need to be output. Figure 16 shows that performance across all data structures when accessing and printing their last element was near identical with times of 0.000 and 0.001 regardless of the size of N. This suggests that for the problem sizes tested, the time complexity is constant- O(1). Therefore, any data structure is a viable option as they can all access their last element equally as fast for the tested sizes of N.

Figure 17 shows the recorded execution times for the test to invert bool values within each data structure. As shown by the the plot and by Figure 14, the bool vector could instantly flip all elements regardless of the size of N and always had a recorded run time of 0.00 (O(1)) seconds, making it the ideal data structure to use if large amounts of bools must be flipped. The list of bitsets was only marginally slower, having an execution time of 0.003 at its max size of 32 bits. Conversely, the boolLinkedList grew quadratically with respect to the size of N, having a time complexity of O(N^2). From this, the bool vector is the best choice for a structure that needs to repeatedly invert all its bool values, but the list of bitsets can be used with a miniscule performance delay. The boolLinklist should be outright avoided for this task.

Finally, figure 18, shows the time to delete the entire data structure as a function of N. As shown by the plot, the deletions of the bool vector and the list of bitsets were linear, although slight jumps are visible in the trace. These jumps are due to large problem sizes being tested and not testing sequential problem sizes closer in size. Meanwhile, the boolLinkedList is shown to only vary slightly with respect to N, having values of 0.00 seconds up to 20480 elements where it changes to 0.001 seconds for all sizes of the integer that were tested. Since these values are so close, the boolLinkedList has a constant time complexity O(1) over the range of problem sizes that were tested. This is largely due to how the boolLinkedList's deletion case was optimally written to repeatedly delete the front node, which saved time as the structure did not have to travel to the furthest node to delete it. It is interesting that this test performed faster than repeatedly executing the list's pop_front function and the vector's pop_back function as all structures now essentially had a direct pointer to the element or node to delete.

From these results, each data structure had a test case where it outperformed the others, but the performance of each data structure across all tests needs to be considered. For instance, the boolVector could instantly invert the value of its elements, but its performance lacked when deleting all its elements and became even worse when outputting all of them. Meanwhile, the list of bitsets could invert its values nearly as fast as the bool vector and output its entire contents much quicker, but had a slower time to delete all of its elements than the boolLinkedList. Finally, the boolLinkedList could delete its elements much quicker than the other tested data structures but its performance suffered when printing and flipping all of its elements. To select the best data structure to hold bool values to solve a problem, the tables and plots shown in figures 14 to 21 can be consulted if the size of the problem, the number of bools to be stored in each node, and the operations that will be performed on the bool values are known.

The space complexity is the second result obtained from this experiment, which determines how the space requires to store the data structure grows as a function of the problem size N. The space complexity for a data structure can be obtained using the formula n(P+E) = D*E, where n is the number of elements in a linked list, P is the size of the pointer to the next node, E is the size of the data stored in the node, and D is the maximum number of elements that can be stored in the array. Hence, the left side of this equation is the size of the linked list and the right is the size of an equivalent array. Usually, the value of n is solved for to determine at what point the array becomes a more efficient solution in terms of the space required to implement it, but the boolLinkedList is inherently more efficient than the array when storing the same amount of data, as proved below by comparing the size required for each structure.

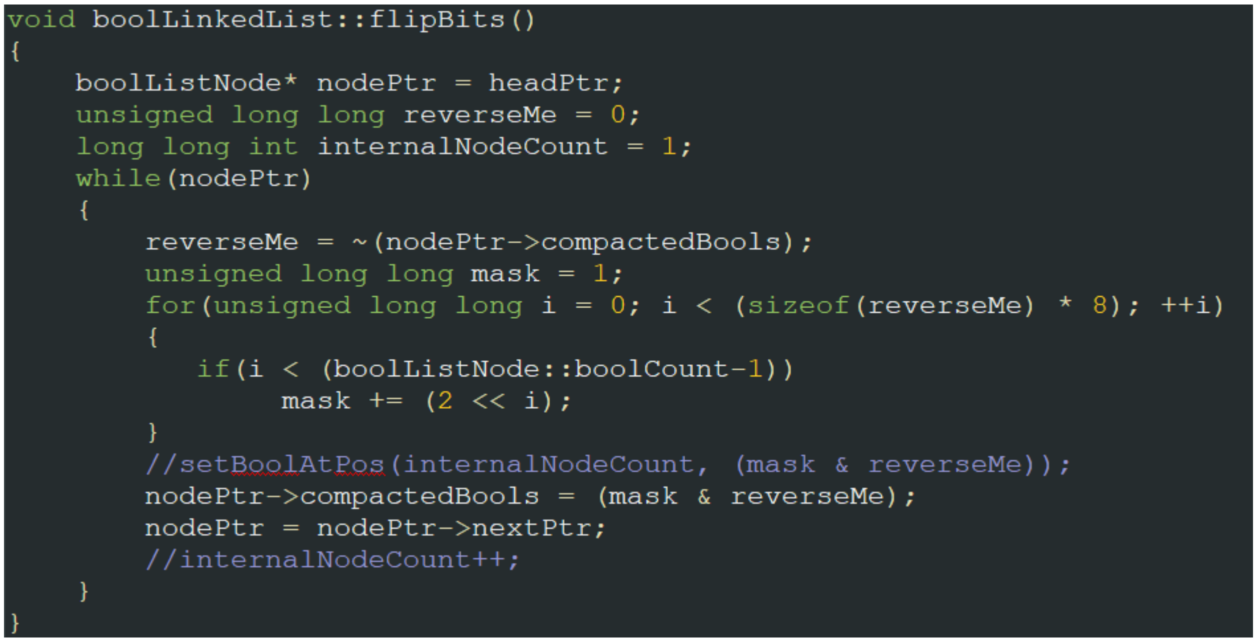
In the boolListNode, the size of the boolListNode pointer is 4 bytes and the size of the data, E_L, can vary between 1, 2, 4, and 8 bytes. Here, n = N, the problem size that the data structure is tasked with. To store an equivalent number of elements in the array, it needs to store 8 values for each byte of the boolListNode's data, so D = n*8*E_LL. The space complexity equation then becomes n(P+E_LL) = n*8*E_LL (E_A is 1 byte). Substituting the various values of E_LL in yields the following:

8 bit: n(4+1) = n*8*1*1 ------> N(5) = N(8)
16 bit: n(4+2) = n*8*2*1 -----> N(6) = N(16)
32 bit: n(4+4) = n*8*4*1 -----> N(8) = N(32)
64 bit: n(4+8) = n*8*8*1 -----> N(12) = N(64)

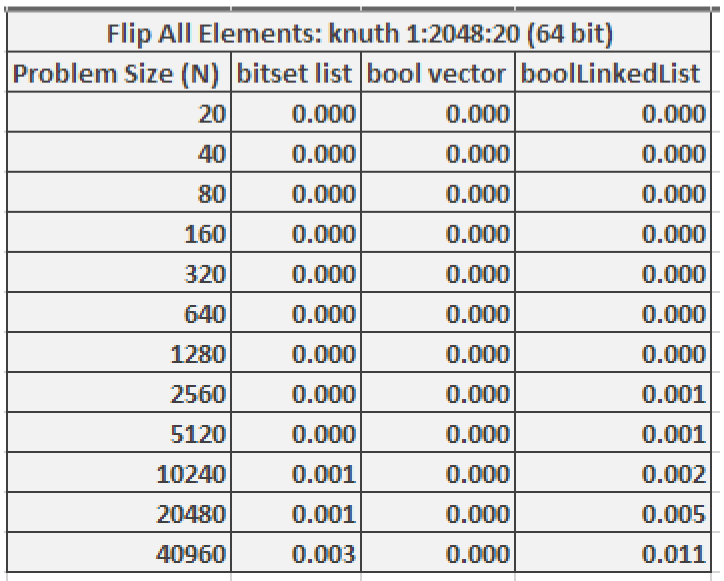
As shown, the boolLinkedList always requires less data than the array to store a number of bool values equal to the size of the problem N, which is the number of nodes in the boolLinkedList. This result is the same for the bitset, as it generates a "fixed-size sequence of N bits" (Std::bitset), which can be 8, 16, 32, or 64 just like the boolLinkedList. Finally, the bool vector is just an array that can grow dynamically and store bools as bits, requiring a size equal to N/8.

With these space sizes in mind, the boolLinkedList and list of bitsets are more efficient with space compared to an array of bools as long as they are storing more bools than the size of their nodes. Thus, the 8 bit boolLinkedList and bitset lists are more efficient when over 5 bools are stored in the array, the 16 bit versions when more than 6 bools are stored, the 32 bit versions when more than 8 bools are stored, and the 64 bit versions if more than 12 bools are stored. Likewise, the bool vector is more efficient than the array of bools by a factor of 8, as it can store the bools as individual bits. Finally, as these sizes all grow as a multiple of the problem size N, each structure has a linear space complexity.

Optimization:
The member functions of the boolLinkedList class dictate how the data structure performs during testing. Poorly written member functions lead to bad performance, as was the case with the flipBits function. In the flipBits function, a call to the theBoolAtPos function was made while the flipeBits function could directly access the value of the node it was currently on. Fixing this error results in the code seen below in Figure 22.


```
void boolLinkedList::flipBits()
{
    boolListNode* nodePtr = headPtr;
    unsigned long long reverseMe = 0;
    long long int internalNodeCount = 1;
    while(nodePtr)
    {
        reverseMe = ~(nodePtr->compactedBools);
        unsigned long long mask = 1;
        for(unsigned long long i = 0; i < (sizeof(reverseMe) * 8); ++i)
        {
           if(i < (boolListNode::boolCount-1))
                mask += (2 << i);
        }
        //setBoolAtPos(internalNodeCount, (mask & reverseMe));
        nodePtr->compactedBools = (mask & reverseMe);
        nodePtr = nodePtr->nextPtr;
        //internalNodeCount++;
    }
}
```
Figure 22: Modified code for flipBits function

This fix removes a call to a function with a time complexity of O(N), making the flipBits function have a linear time complexity as well => O(N). Performing the flipping all values test on the 64 bit application of the boolLinkedList with this change made produced the run times shown in figure 23.



Flip All Elements: knuth 1:2048:20 (64 bit)			
Problem Size (N)	bitset list	bool vector	boolLinkedList
20	0.000	0.000	0.000
40	0.000	0.000	0.000
80	0.000	0.000	0.000
160	0.000	0.000	0.000
320	0.000	0.000	0.000
640	0.000	0.000	0.000
1280	0.000	0.000	0.000
2560	0.000	0.000	0.001
5120	0.000	0.000	0.001
10240	0.001	0.000	0.002
20480	0.001	0.000	0.005
40960	0.003	0.000	0.011

Figure 23: Runtime results of the flip all elements performed on each data structure storing 64 bits, with the boolLinkedList's optimized flipBits function.

Thus, the boolLinkedList is now much closer in execution time to the other 2 structures for this test, boosting its overall performance and efficiency.

Conclusion:

In conclusion, it was determined that it is possible to write a custom data structure that can manipulate stored integers to represent collections of bool values equivalent to the size of the integer in bits. The boolLinkedList class did this and utilized member functions that provided the illusion of bit access and manipulation on any bit within the integer. Bit operations were used to facilitate this, with the impression that they could be quickly performed and result in faster performance than the same functions on the STL structures. This was sort of the case, as the boolinkedList's test results often fell in between those of the STL structures. The variances in time between all structures were small except for the test that required all of the bools within the structure to be flipped, at which point the boolLinkedList began taking much longer. This was due to a poorly written flip bits function that had a time complexity of O(N^2). Optimizing the flipBits function as discussed previously brought its time complexity down to O(N), bringing the boolLinkedList must closer to the STL structures in terms of performance. This serves as a reminder of how poorly written algorithms ran on a data structure result in the appearance of bad performance, and how important it is to write these algorithms efficiently.

As shown by the results of this research, the boolLinkedList is necessarily more efficient than STL structures such as a list of bitsets or a bool vector, as those structures generally had faster execution times when tested compared to the custom class. However, for the problem sizes tested, the tables and plots shown in figures 12 through 21 show that the data structures only had small differences in runtimes with respect to the tests performed. Smaller difference in runtimes were experienced with smaller values of N, while larger values of N resulted in larger variations.

Concerning the space complexity of the boolLinkedList, it becomes a more space efficient than an array once a greater number of bools are stored than the cost of a single node. This means that the boolLinkedList's 8 bit form is more efficient if more than 5 bools are to be stored, the 16 bit form if more than 6 bools are to be stored, the 32 bit form if more than 8 bools are to be stored, and the 64 bit form if more than 12 bools are to be stored. Since the amount of space needed by each node is constant, the space complexity grows linearly with respect to the problem size N. Additionally, the boolLinkedList becomes more space efficient in its larger forms, with the 64 bit form requiring 3/16ths of the space that the equivalent bool array would need. However, this trait is shared by the list of bitsets, and neither the boolLinkedList or list of bitsets are more space efficient than the boolVector.

Therefore, the boolLinkedList does not necessarily outperform the tested structures with respect to the problem size N and the bool vector remains a more space efficient solution. Due to this, is there ever an instance where this class should be used, or another similar class should be written? The answer is yes, as this class trades performance for increased functionality. The boolLinkedList allows member functions to be implemented that can manipulate a standard type in ways not supported by the STl data structures, or ways in which require the programmer to think very creatively. Thus, the boolLinkedList can implement this additional functionality with a slight time penalty. Therefore, the boolLinkedList is the optimal solution where manipulations need ot be performed on bool values that the STL does not support, or when the problem size is so small that no performance penalty is incurred.

References:

Crissey Renwald, W. (n.d.). *U2-M3 Linked List* [PDF]. Akron: University of Akron Department of Computer Science.

Obtained from taking the CSII Course

Rand_max. (n.d.). Retrieved February 07, 2021, from https://en.cppreference.com/w/cpp/numeric/random/RAND_MAX

Roberts, E. (2015, January 20). *Memory and C++*. Lecture presented at CS 106B. Retrieved February 07, 2021, from http://stanford.edu/class/archive/cs/cs106b/cs106b.1154/handouts/21-MemoryAndC++.pdf

Std::bitset. (n.d.). Retrieved February 07, 2021, from https://en.cppreference.com/w/cpp/utility/bitset

Appendix A

<u>boolLinkedList Member Function Summary</u>

01. **boolLinkedList()**
    a. Default constructor which created an empty boolLinkedList
02. **boolLinkedList(std::initializer_list<int> boolList)**
    a. Default constructor that initializes boolLinkedList with integer elements from an initialization list.
03. **~boolLinkedList();**
    a. Destructor to destroy boolLinkedList
04. **addBoolListNode(unsigned long long int boolRep)**
    a. Appends a new boolListNode to the end of the linked list
05. **void insertBoolListNode(unsigned long long int nodePlace, unsigned long long int boolRep)**
    a. Inserts a new boolListNode at the specified position
06. **deleteBoolListNode(unsigned long long int nodePos)**
    a. Deleted boolListNode at specified position
07. **unsigned long long int getNodeCount();**
    a. Return nodeCount
08. **void printBools();**
    a. Print value of all nodes in boolLinkedList to console
09. **unsigned long long int getBool(unsigned long long int nodeIndex)**
    a. Return value of compactedBools at specified position
10. **int bitShiftCompactedBool(unsigned long long int compactedBool, int boolPosition)**
    a. Returns bool at specified location within compactedBools
11. **int isolateBool(unsigned long long int, int);**
    a. Returns bool at specified location in specified binaryListNode
12. **setBoolAtPos(unsigned long long int nodeIndex, unsigned long long int newBool)**
    a. Change value of compactedBools at specific binaryListNode
13. **void modifyBoolAtPos(unsigned long long int nodeIndex, int boolPosition, bool newBool)**
    a. Change value of specified bool at specified position in specified binaryListNode
14. **void recursiveBitShift(unsigned long long int bitPosMask, unsigned long long int compactedBool)**
    a. Uses recursion to print binary equivalent of compactedBools
15. **void displayNodeBinary(unsigned long long int nodeIndex)**
    a. Display binary equivalent of compactedBools in specified node
16. **void displayNodeBinaryFormatted(unsigned long long int nodeIndex)**
    a. Display formatted version of binary equivalent of compactedBools in specified node

17. **void displayListBinary();**
    a. Display binary equivalent of compactedBools in entire boolLinkedList
18. **void displayListBinaryFormatted();**
    a. Display formatted binary equivalent of compactedBools in entire boolLinkedList
19. **void flipBits();**
    a. Invert bools in each boolListNode in boolLinkedList