C++ Sparse Array Container

# Lamya Bhasin.

## PES1201701244

*Department of Computer Science and Engineering*

*PES University*

Bangalore [lamyabhasin01](mailto:zenkarsrinivas@gmail.com)[@gmail.com](mailto:as@gmail.com)

# Kumar N S

## Professor

*Department of Computer Science and Engineering PES University*

Bangalore

[kumaradhara@gmail.com](mailto:kumaradhara@gmail.com)

***Abstract*—Sparse array is an array of data n which many elements have the value zero. In this project we have implemented a conceptual array called SparseArray, which is a container, along with some functions, algorithms and its iterators.**

***Index Terms*—Containers, Iterators, Algorithms**

1. INTRODUCTION

Linear data structures like arrays have significant applications in fields of study like image processing, computer graphics and various complex algorithms. Efficient processing of these arrays is vital for algorithms used in these fields. When it comes to sparse arrays it becomes even more necessary that unnecessary processing of the indexes with null values is avoided. Keeping this in mind a container and related efficient operations are proposed through this paper. The project is an attempt to implement a sparse array container to support efficient storage and operations for a one-dimensional array. The container makes use of C++ generics to support array storage using a doubly linked list. The user interacts with the container as though it were a regular random-access array, but is internally interfacing with a doubly linked list. The generic nature of the container allows it to store data of any primitive or user defined data type. This project also implements iterators and sort function for this sparse array container. All operations and functions written are done in optimal time.

# Anish Poddar

## PES1201700269

*Department of Computer Science and Engineering*

*PES University* Bangalore anishpoddar2307[@gmail.com](mailto:endras2@gmail.com)

1. IMPLEMENTATION

The project implemented using C++ tries to make the best use of the template classes and template functions to make the container generic without compromising on the efficiency of operations. The container and the node stored are defined as a template.

template *<*typename T*>* class SparseArray

The main container as a member contains the size of the conceptual array and the pointers-head and tail to the doubly linked list. The node of the list is an object of template class SparseElement which containers members like index of the element in the conceptual array, the element and the next and previous pointers to the object of this class. By just storing the non-zero elements we reduce the space complexity of storing to theta (number of non-zero elements). Defined within the scope of the container is an Iterator about which we will discuss in the Iterator section.

1. *Sparse Array Storage Class*

The Sparse array, as mentioned earlier, is implemented as a doubly linked list. This class contains the head, tail and size of this list. It also contains a constructor to initialize these members with their default values. We have also overloaded the ‘[]’ operator to return the r-value and l-value so that when a[index1] = a[index2] is performed the correct aim is achieved. If an index is not initialized then we create another node and insert it to the list. If an index already exists then we only overwrite its element value.

1. *Operations*

The implemented container supports the following opera- tions:

* + Copy Assignment : Performs a deep copy and returns reference to the assigned container
  + begin(): returns an iterator pointing to first non-zero element of sparse array
  + end(): Returns an Iterator to end of sparse array i.e nullptr as it is actually a list.
  + Indexing operator[] : Returns the r-value or l-value depending on its usage.
  + Element Insertion: Inserting non-zero element at specified index.
  + Sort: sorts the array in ascending order.
  + ++(Iterator): gives iterator to the next element in the array.
  + (Iterator)++: gives iterator to next element by pre-incrementing.
  + --(Iterator): gives iterator to the previous element in the array.
  + (Iterator)--: gives iterator to previous element by pre-incrementing.
  + \*: dereferencing an iterator return the element that the iterator points to.
  + ==(iterator): checks if two iterators are equal to each other.
  + !=(iterator): checks if an iterator is not equal to another iterator. It returns true or false.
  + <<(complex): has been overloaded for outputting complex numbers
  + >>(complex): has been overloaded to input complex numbers
  + ==(complex): has been overloaded to check if 2 complex numbers are equal

1. *Iterator*

The Container supports Iterator for effective traversal through non-zero elements of the matrix. The Iterator belongs to forward Iterator Category. The Iterator is compliant with standard algorithms find and accumulate which are applicable for an array. The Iterator class is implemented as a wrapper around the Iterator of the sparse array storage class. It contains as its members a pointer to the sparse array. The Iterator performs a linear traversal over the non-zero elements of the sparse array. The Iterator supports two constructor overloads.

The Iterator supports the following operations:

* + Equality: Accepts an Iterator and returns true if both point to same element.
  + Inequality: Accepts an Iterator and returns false if both point to same element.
  + Dereferencing: Returns a reference to the data at current position of the iterator.
  + Pre-increment: Returns reference an Iterator to next element’s node
  + as pre-increment, except it returns r-value instead of l-value.
  + Pre-decrement: Returns reference an Iterator to previous element’s node
  + Post-decrement: Same as pre-decrement, except it returns r-value instead of l-value.

1. RESULTS

From the point of view of generics, the proposed container supports primitive and user-defined types and operations like searching, sorting etc which can be performed on any array irrespective of container element type used.

Very sparse arrays(15% dense) hog up a lot of mem- ory(about 85% is wasted) and take the same amount of time for operations, despite being mostly populated by elements that do not contribute to the result. For such arrays, our container takes up only memory for the non-zero elements, and performs faster operations.

Time complexity:

Sorting – O(n2)

Search - O(m)

Random access using [] – O(m)

Display – O(n)

Where, n is the size of the sparse array and m is the number of elements currently in the array.

1. CONCLUSIONS

The container is an effective representation way for saving memory besides boosting the algorithms. The drawback of this is when the number of elements added is close to the total capacity or size of the array then the efficiency decreases. The sparse array can be supported to store data of various types by making use of generic features of C++ such as template classes, template functions, friend functions, etc.

ADDITIONAL WORK

The concept of the SparseArray may be extended to a sparse matrix in multiple dimensions, where each of the coordinates of its elements would be stored as a separate field in the SparseElement object. This would require iterators for traversing the array in all possible directions (e.g. horizontally and vertically for a 2D matrix).

ACKNOWLEDGMENT

The members of this project would like to thank Prof. Kumar NS for providing us an opportunity to work on this interesting project which helped us explore various features of C++. Through this project we were able to experiment different designs for the Sparse array container by thinking not only in terms of its generic usage but also in terms of efficiency of the related operations in terms of space and time.

REFERENCES

* [https://www.cplusplus.com/](http://www.cplusplus.com/)
* https://en.cppreference.com/w/
* [https://www.wikipedia.or](http://www.wikipedia.org/)g/
* [https://www.geeksfor](http://www.geeksforgeeks.org/)geeks.or[g/](http://www.geeksforgeeks.org/)