Chapter 7

Data Structure

Data structure is one of the main parts of a finite element program. Many restrictions in functionality of finite element codes comes from their data structure design. The Kratos' data structure has to provide the high flexibility, necessary for dealing with multi-disciplinary problems.

In this chapter first, a brief description of different concepts in data structure programming is given. Then, some classical containers are explained and their advantages and disadvantages are discussed. It continues with design and implementation of new containers suitable for multi-disciplinary finite element programming. After that, some common organization of data in finite element programming is explained and finally the organization of the data structure in Kratos is described.

7.1 Concepts

In this section a brief description of common concepts in data structure programming is given. More information about the concepts described here and also other concepts in this field can be found in [55, 14, 46].

7.1.1 Container

Container is an object which stores another objects and gives some method to access these objects, add new objects, or remove some objects from it. Each object stored in a container is referred as an *element* of it. Container must provide some methods for creating and modifying it and also some access method to its element. Here is a list of some common methods:

Access Gives the element in given position. Depending on the container the access may be implemented via position or by some reference key. Usually the `[]` operator provides this interface for container.

Insert Inserts a given element after given position. Adding elements to a container increases its size.

Append Adds given element to the end. Adding elements to a container increases its size.

Erase Removes the element in given position. This operation reduce the size of container by the number of erased elements.

Find Searches for an element with given specification in container.

Size To get size of the container.

Resize Changes the size of container.

Swap Swaps the content of the container with a given one.

Some containers may not provide all these methods due to their structure and some other may provide some more methods for their specific uses. Also the performance of these operations depends highly on the internal structure of container. So before using a container it is very important to study its performance in term of the operations needed by algorithm.

7.1.2 Iterator

Usually a pointer referred as an *iterator* is used to access elements of a container. Here is an example of using a pointer as a C array iterator to print its contents:

```
double data[10];

// putting values in data
// ...

// now printing data using an iterator
double* data_end = data + 10;
for(double* i = data ; i != data_end ; i++)
    std::cout << *i << std::endl;
```

The *Iterator* pattern defines a generalized pointer to access elements of a container sequentially without exposing its internal structure. An iterator can be used to traverse element by element the container in a general way and without knowing how they are really stored in memory. Changing the C array of previous example to a container gives an example of using iterator in a general form:

```
ContainerType data;

// putting values in data
// ...

// now printing data using an iterator
typedef ContainerType::iterator iterator_type;
 for(iterator_type i = data.begin() ; i != data.end() ; i++)
    std::cout << *i << std::endl;
```

In this example any container which provides an iterator and two methods to indicate its begin and end position can be used to print its contents.

Iterator can be designed to traverse the container in different manner. For example a matrix can have different iterators:

`iterator` iterates over all members from a_{11} to a_{mn}

`row_iterator` iterates over rows of the matrix.

`column_iterator` iterates over columns of the matrix.

`nonzero_iterator` iterates over all nonzero element.

This makes iterator a very powerful tool to access a container in a generic way. For algorithms using iterators there is no need to know about the container itself and this make them more generic.

Depending on the structure of container there are certain traversing is impossible or meaningless. So different containers use different category of iterators due to their restrictions. Here are the main categories of iterators:

Forward Iterator A simple iterator which allows just moving to the next element. This iterator cannot be used to go backward. For example to see the previous element of iterator position. Forward iterator can move only one element forward in each step and cannot be used to jump by a certain offset.

Bidirectional Iterator Unlike forward iterator this iterator can be used to traverse back the container. But still can move on step forward and backward each time.

Random Access Iterator This iterator can move freely forward and backward and also jump to any other position given by an offset.

7.1.3 List

List is a sequence of elements that can be linearly ordered according to their position on the list. A list is usually represented by a comma separated sequence of element:

$$a_1, a_2, ..., a_{i-1}, a_i, a_{i+1}, ..., a_n$$

What is important in a list is the relative position of elements. All the elements are in the same level and the only relation between them is to be the next element or previous one. For example the element a_{i-1} is before a_i and a_{i+1} is the next one.

There are several structure representing a list. Each of them store its elements in a different way and provides different properties as we will see later.

7.1.4 Tree

Tree is a hierarchical collection of elements referred as *nodes*. Unlike the list the elements of the tree are not in the same level and some of them considered to be the parent of some other. In each tree there is a node called *root* which is the parent of whole tree.

7.1.5 Homogeneous Container

A container which is able to store only one type of elements is a *homogeneous container*. For example a C array of doubles is homogeneous while can store only double variables.

```
double homogeneous_array_of_doubles[3];
```

Even an integer must be converted first to double and then it can be stored in this array. The C++ standard containers are homogeneous and can store only one type of elements.

7.1.6 Heterogeneous Container

A container considered to be *heterogeneous* if is able to store different types of elements. Figure 7.1 shows an heterogeneous container in memory.

Usually a heterogeneous container is implemented to accept any type of data but in some cases its more convenient to implement a container which is also heterogeneous but can only store

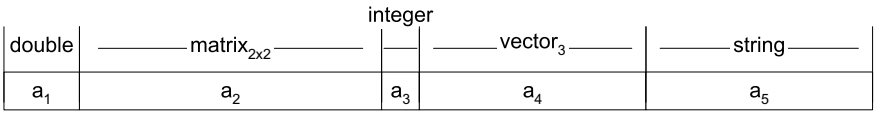


Figure 7.1: An heterogeneous container in memory storing a double, a matrix, an integer, a vector and also an string.

certain types of elements. In this work this type of containers will be referred as *quasi heterogeneous containers*.

7.2 Classical Data Containers

In this section some classical data containers which are usually used in finite element programs are briefly described and their advantage and disadvantages are also discussed. Further information about data containers and their implementation can be found in [55, 14, 46, 102].

7.2.1 Static Array

Static array is an implementation of a list which puts its elements sequentially in memory without any gap between them. Figure 7.2 shows an array in memory.

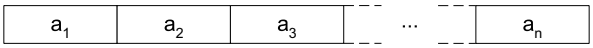


Figure 7.2: Array elements sequentially stored in memory

Static array can store certain number of element given in time of construction. For example the following array of doubles can hold up to 10 doubles:

```
double results[10];
```

This restriction makes static array unusable for variable size and growing containers.

Interface and Operations

The interface of an array is very simple due to its restricted functionality.

Access Usually, operator `[]` is used to access elements of an array. Accessing is very fast and is constant time respect to the position and also size of the array. This means that the time to access any element in an array is not dependent on its position and neither on the number of elements in container. Accessing an element is done by offsetting the base pointer by position index as shown in figure 7.3.

Size To provide size interface the array must store its size which implies an overhead specially in the case of small arrays. For some implementations this overhead can be eliminated as we will see later.

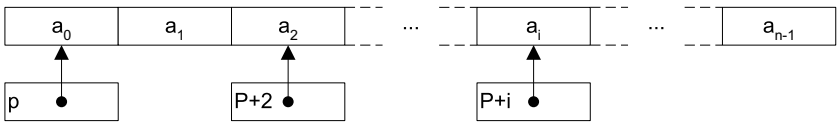


Figure 7.3: Array indexing

Swap In general swapping can be down element by element which causes swapping time to be linear respect to the size of the container. In some cases this can be down via pointers which make it a constant time operation because two arrays just change their pointing sequence and not all elements one by one.

As mentioned before the size of an array cannot be changed so there is no resize interface for the static array. For same reason there are no insert, append and erase interfaces while these interfaces will change the size of container.

Advantages

- Very good use of system cache. The sequential nature of array makes it very cache efficient which can make a large increase in its performance.
- Iterating over an array is very fast. In any moment the next and previous elements are known and using cache memory makes iteration extremely fast.
- Each element is accessible very fast by its sequence number because knowing this number is enough to know its position without iterating over array.
- loop over elements of an static array can be optimized more for small arrays by unrolling the loop while the number of elements can be known in compilation time.
- No memory overhead per element. In some implementations even no memory overhead per container. This makes it a very good choice when a large amount of small containers is needed.

Disadvantages

- There are no way to insert a new element or erase an exiting one. This makes it unsuitable for variable size sequences.
- The size of an array must be known in time of creation which makes it unusable for growing sequences of elements.

In general static is well suited for small and rigid containers. For example a 3 dimensional point can be implemented as an static array of 3 elements.

Implementation

C (and consequently C++) provides an static array by itself. This array is the minimum implementation of an array which provides a pointer to iterate over it and a `[]` operator to access its data. C array do not provide size information which implies users to provide this additional information to any algorithms they pass a C array.

The simple C array can be improved using C++ templates as follows:

```
template<class TDataType, std::size_t TSize>
class array
{
    TDataType mElements[TSize];
  public:

    // type definitions
    typedef T              value_type;
    typedef T*             iterator;
    typedef const T*       const_iterator;
    typedef T&             reference;
    typedef const T&       const_reference;
    typedef std::size_t    size_type;
    typedef std::ptrdiff_t difference_type;

    // iterator support
    iterator begin() { return mElements; }
    const_iterator begin() const { return mElements; }
    iterator end() { return mElements+TSize; }
    const_iterator end() const { return mElements+TSize; }

    // access operator[]
    reference operator[](size_type i)
    { return mElements[i]; }
    reference operator[](size_type i) const
    { return mElements[i]; }

    static size_type size() { return TSize; }

    // element by element swap (linear complexity)
    void swap (array<T,TSize>& y)
    {
        std::swap_ranges(begin(),end(),y.begin());
    }

    // c array representation
    T* c_array() { return mElements; }
};
```

This implementation provides the size of array without any storage overhead. Also it is STL compatible and can be passed to STL algorithms which gives another added value to it. In this project the boost [3] implementation of the array is used which announced to be a part C++ standard draft.

7.2.2 Dynamic Array

There are many situations in which the exact size of array is not known but a maximum size can be given. This maximum size can be used as capacity of the array. So the array can be constructed with a given capacity and used to store any number of elements less than the capacity. Here an additional pointer, pointing to the tail of stored elements, is necessary to indicate the actual size of the array. Figure 7.4 shows this implementation of the array.

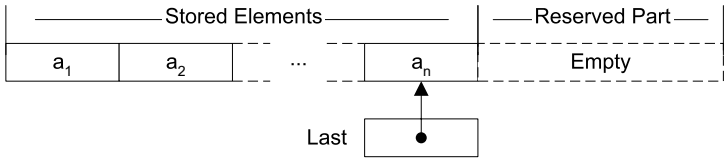


Figure 7.4: Array constructed with given capacity

Interface and Operations

Access Usually, operator `[]` is used to access elements of an array. Like static array accessing is very fast and is constant in time. This means that the time to access any element in an array is not dependent on its position neither on the number of elements in the container. Accessing an element is done by offsetting the base pointer by position index starting from zero as shown in figure 7.5.

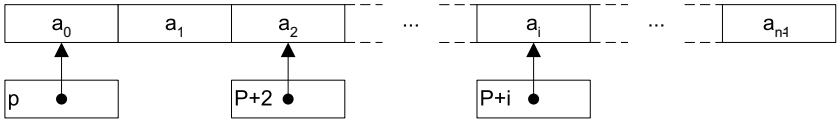


Figure 7.5: Array indexing

Insert Inserting an element in a position can be done by shifting all elements after that position to their next position in order to make room for a new element and then put the element in the prepared position. This operation is linear respect to the number of elements to be shifted. Figure 7.6 shows this procedure.

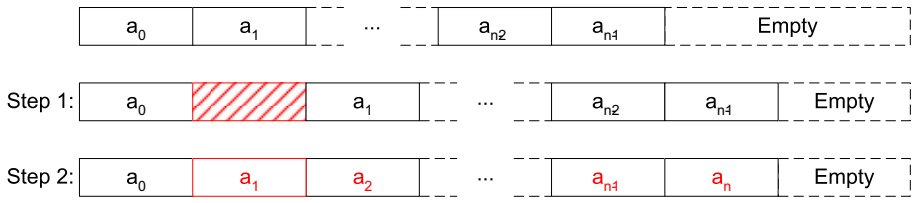


Figure 7.6: Adding an element in second position which causes all the rest of elements to move one step forward to make room for new one

Obviously this procedure is valid until the capacity of the array is more than its size. Otherwise a resize procedure is required.

Append Unlike inserting, appending an element to the end of array do not need any shifting and takes constant time independent of the size of array. Figure 7.7 shows this procedure. Again if the array is full append causes resizing with capacity changing which make it less efficient.

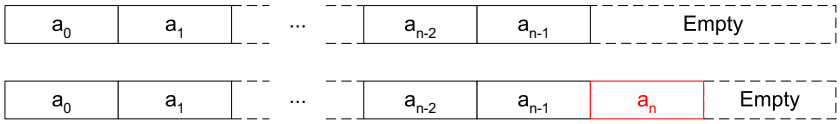


Figure 7.7: Appending an element to the end of array

Erase To erase an element in the middle of array again a shifting process is necessary to reconstruct the continuity of array. Erasing consist of removing the element and then shift the rest of element one step back to fill the gap. Figure 7.8 shows this procedure.

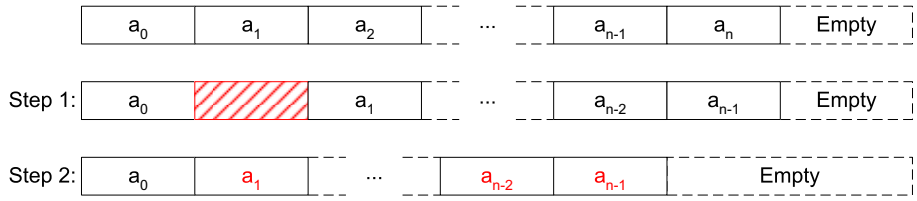


Figure 7.8: Erasing the second element of array

Size The size of the array can be calculated by the distance of begin and end of array or it can be stored depending on the implementation. In general it is a fast process which is constant in time respect to the size of container.

Resize Resizing without changing capacity is simple and efficient but changing the capacity is more complex and inefficient. Changing the capacity of an array implies reallocation of memory. If there is free memory after this array reallocation can be done without copying but if the memory is allocated then whole array must be copied to some other places with sufficient space as shown in figure 7.9. This causes the pointers to elements of array to be invalidated while they are still pointing to the previous position of the array in memory.

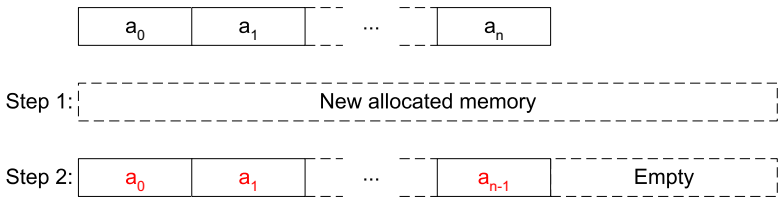


Figure 7.9: Resizing an array beyond its capacity when there is no space after array to grow

Swap In general swapping can be down element by element which causes swapping time to be linear respect to the size of the container. But in some cases this can be down via pointers which make it a constant time operation because two arrays just change their pointing sequence and not all elements one by one.

Array Advantages

- Very good use of system cache. The sequential nature of array makes it very cache efficient which can make a large increase in its performance.
- Iterating over an array is very fast. In any moment the next and previous elements are known and using cache memory makes iteration extremely fast.
- Each element is accessible very fast by its sequence number because knowing this number is enough to know its position without iterating over array.
- No memory overhead per element. There is just an small overhead of storing to pointer for each container.

Array Disadvantages

- Inserting elements in the middle of array is relatively slow due to the shifting procedure.
- Increasing the capacity of vector requires reallocating of memory which makes it slow. Reserving extra memory solves this problem with the cost of memory overhead.
- Shifting and reallocating invalidate pointers to the elements of an array, which makes element referencing a difficult task. Though using vector with sequence number instead of direct referencing can solve this problem.
- Array do not reduce the capacity automatically. For example removing elements from the array will not reduce the memory used by the array.

Interface and Operations

Access Usually, the operator `[]` is used to access elements of an array. Accessing is very fast and is constant time respect to the position and also size of the array. This means that the time to access any element in an array is not dependent on its position neither to the number of elements in container. Accessing an element is done by offsetting the base pointer by position index starting from zero as shown in figure 7.10.

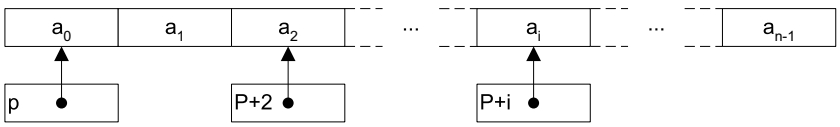


Figure 7.10: Array indexing

Insert Inserting an element in a position can be done by shifting all elements after that position to their next position in order to make room for new element and then put element in prepared position. This operation is linear respect to the number of elements to be shifted. Figure 7.11 shows this procedure.

Obviously this procedure is valid until capacity of array is more than the size. Otherwise a resize procedure is required.

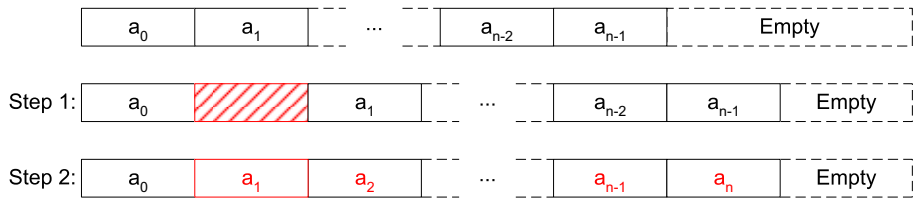


Figure 7.11: Adding an element in second position which causes all the rest of elements to move one step forward to make room for new one

Append Unlike inserting, appending an element to the end of array do not need any shifting and takes constant time independent of the size of array. Figure 7.12 shows this procedure. Again if the array is full, append causes resizing with capacity changing which make it less efficient.

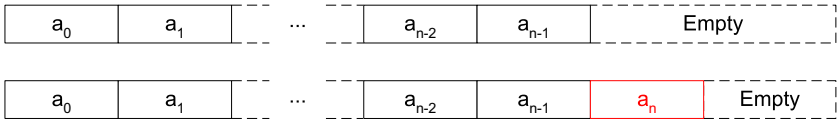


Figure 7.12: Appending an element to the end of array

Erase To erase an element in the middle of array again a shifting process is necessary to reconstruct the continuity of array. Erasing consist of removing the element and then shift the rest of element one step back to fill the gap. Figure 7.13 shows this procedure.

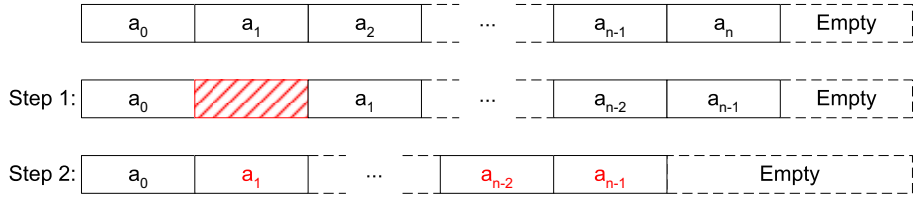


Figure 7.13: Erasing the second element of array

Size The size of array can be calculated by distance of begin and end of array or it can be stored depending on implementation. In general its a fast process with constant time respect to the size of container.

Resize Resizing without changing capacity is simple and efficient but changing the capacity is more complex and inefficient. Changing the capacity of an array implies reallocation of memory. If there is free memory after this array reallocation can be done without copying but if the memory is allocated then whole array must be copied to some other places with sufficient space as shown in figure 7.14. This causes the pointers to elements of array to be invalidated while they are still pointing to the previous position of the array in memory.

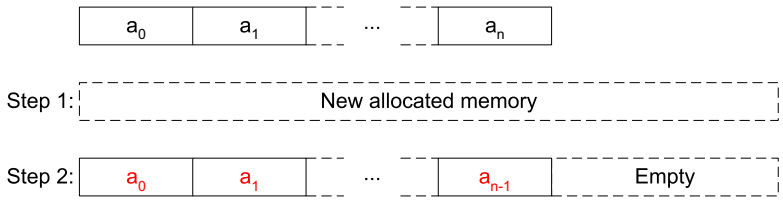


Figure 7.14: Resizing an array beyond its capacity when there is no space after array to grow

Swap In general swapping can be down element by element which causes swapping time to be linear respect to the size of the container. But in some cases this can be down via pointers which make it a constant time operation because two arrays just change their pointing sequence and not all elements one by one.

Implementation

As mentioned before new elements can be appended to the tail of list very easily but inserting an element in the middle of the list requires all following elements to shift and make room for new one. Also removing elements from tail also is easy while removing from the middle is more complex due to the shifting procedure of following elements to close the gap. The shifting procedure causes inserting and removing elements in the middle of the list to be slower. It also invalidates all the pointers to the shifted element. This causes problem in accessing array elements via pointers. For example the following code for removing the negative values from given data will not work because removing each element changes the position of last element and so the end of sequence.

```
double* begin = data.begin();
double* end = data.end(); // Will be invalidated by remove!!

for(double* i = begin ; i != end ; i++)
   if(*i < 0.00)
       data.remove(i); // Invalidates the end pointer!!
```

Adding any element to an already full array requires increasing the capacity. This operation is slow and doing it for every added element over capacity, results poor performance. Reserving more to be used later is a common approach to this problem. Each time an increasing is needed an additional memory is allocated to avoid reallocation for next elements. Though this approach effectively solves the performance problem but applies a memory overhead. Larger buffer provides better performance and also larger memory overhead. All this can be avoided by creating an array with correct capacity.

`vector` class provides the array implementation in C++ standard library.

```
vector<class T, class Alloc = Allocator<T> >
```

The first template parameter `T` is the type of element to be store and second is the allocator which manages the memory used by vector. `vector` adjust its capacity automatically and also reserves an extra memory to improve the oversize inserting performance. The buffer size varies from one implementation to other but usually is 50% or 100% of vector size. This may cause unacceptable overhead for some cases and must be avoided by assigning correct capacity to the vector.

7.2.3 Singly Linked List

Singly linked list is a set of elements chained to each other by pointer. The main idea is giving each element a pointer to the next one. So having the first element the list can be traversed by going from each element to the next known position. Figure 7.15 shows a singly linked list in memory.

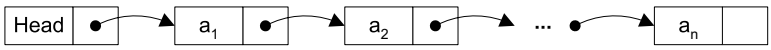


Figure 7.15: Singly linked list stored individually each element with a link to the next element

Interface and Operations

Access Accessing an element in singly linked list knowing its sequence number is not as easy as array. For accessing an element one must start from head of the list and go through the list to arrive to given position. For example to find fifth element of the list first the head must be used to find the first element, then first element have an associated pointer to the second one. Going to second one we can get the pointer to third element and from third to forth and finally to the fifth position. This searching nature makes indexing in a singly linked list a very slow procedure. Some implementations even do not provide access by position.

Insert Inserting a new element after given element is fast with no need to shifting. The inserting procedure consist of making new element pointing to the element the given one is already pointing and making the given element pointing to new one as its next element. Figure 7.16 shows this procedure.

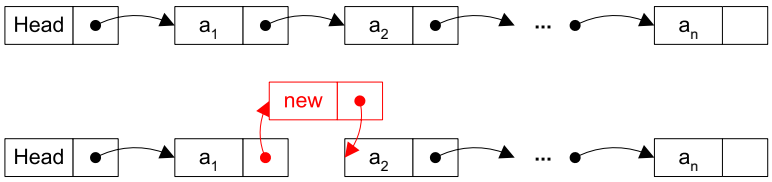


Figure 7.16: Inserting a new element after first element of the list

Append For a linked list append and insert has the same nature and can be done in the same way. The only difference is that there is no pointing element at the end so appending is to make the last element point to the new one as can be seen in figure 7.17.

Erase Like inserting a new element erasing an existing one is very simple and efficient. To erase an element after a given one is enough to make given element pointing to the element after the removing one and then delete it from memory as shown in figure 7.18. The time complexity of erasing an element is constant respect to its position and also to the size of array.

Size Getting the size of a linked list is not as easy as an array. In fact to find out how many element are connected together a complete traverse of list is necessary. So its convenient to accept the overhead and store the size of list for increasing its performance.

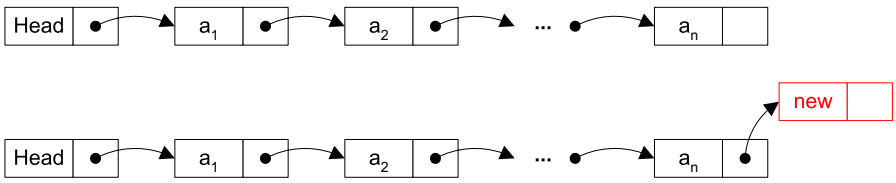


Figure 7.17: Appending a new element after last element of the list

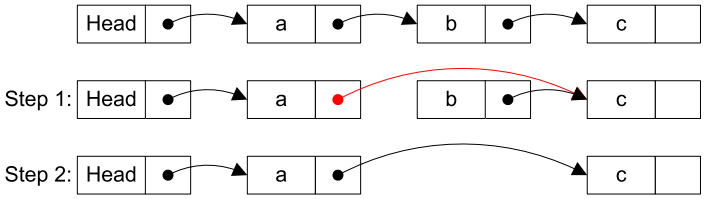


Figure 7.18: Erasing an element from list

Resize While a list do not have a maximum size or any restricted size, this process consist of adding new element or erase some others to achieve given size of the list.

Swap Swapping two lists is very simple and consists of swapping only the head pointer of the two lists. Figure 7.19 shows this procedure.

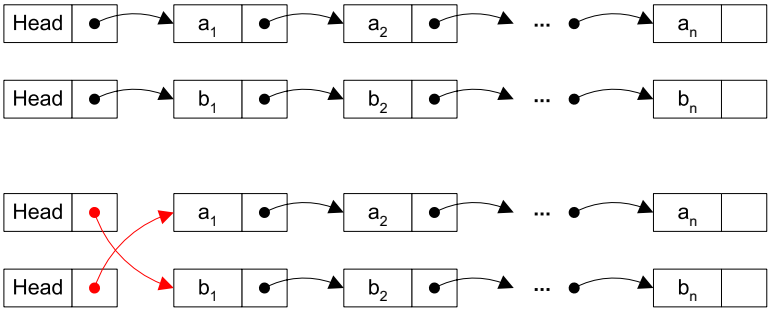


Figure 7.19: Swapping two list by changing their head pointer

Singly Linked List Advantages

- Fast inserting a new element after a given one.
- Efficient removing of an element.
- A linked list can grow smoothly by adding new elements and without the need to resize it or creating a buffer like dynamic arrays.

- Removing elements from lists automatically reduces its size in memory which avoids the corresponding overhead or the manual controlling of its size.

Singly Linked List Disadvantages

- Singly linked list cannot be traversed in reverse direction because each element only knows its next neighbor but not the previous one. So this container cannot be used for algorithms which needs to traverse forward and backward the container.
- Accessing by position is a very slow task and has to be avoided. This accessing can be reduced by keeping the pointer to the necessary elements of the list for future use.
- Lists are used to be less cache efficient due to the fact that its elements can be arbitrary distributed in the memory. This makes them less attractive as fast iterating containers in practice.

A singly linked list is therefore suitable for highly variable data structure subjected to procedures with a lot of element inserting and deleting statements.

Implementation

Unfortunately C++ standard library do not support this container. There are some extensions to standard the library which provide `slist` [28, 8] as singly linked list variant of the standard list container.

In Kratos a modified version of a singly linked list is implemented in the `ProcessInfo` class to keep track of its history during a finite element procedure.

7.2.4 Doubly Linked List

As mentioned above a singly linked list has the drawback that cannot be traversed in reverse order. Doubly linked list solves this problem by adding another pointer to its elements which links them to their previous elements. In this way a doubly linked list can be traversed in both ways but causes an overhead of one pointer per each element. Figure 7.20 shows a doubly link list in memory.

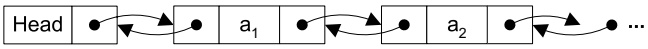


Figure 7.20: A doubly linked list in memory

Interface and Operations

Access Accessing an element in a doubly linked list knowing its sequence number requires the same procedure as for the singly linked list one. Again for accessing an element one must start from the head of the list and go through the list to arrive to the given position. For example to find the fifth element of the list first the head must be used to find the first element, then the first element has an associated pointer to the second one. Going to the second one we can get the pointer to the third element and from the third to the forth and finally to the fifth position. This searching nature makes indexing in a doubly linked list a very slow procedure. Some implementations even do not provide access by position.

Insert Inserting a new element after given element is fast with no need to shifting. It consists of linking the element in the given position to the new one and also the new one to the next element of the given position. Figure 7.21 shows this procedure.

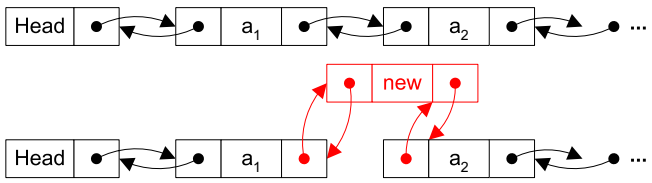


Figure 7.21: Inserting a new element after first element of the list

Append For a linked list append and insert has the same nature and can be done in the same way. The only difference is that there is no pointing element at the end so appending is just making the last element point to the new one as shown in figure 7.22.

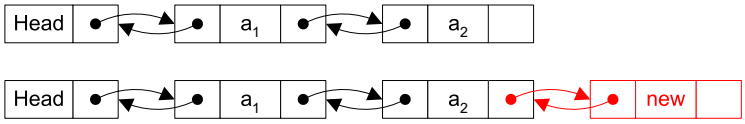


Figure 7.22: Appending a new element after last element of the list

Erase Like inserting a new element erasing an existing one is very simple and efficient. To erase an element after a given one is enough to make the given element pointing to the element after the removing one and then delete it from memory as shown in figure 7.23. The time complexity of erasing an element is constant respect to its position and also to the size of the array.

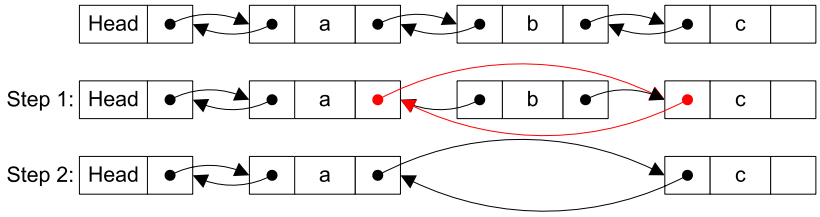


Figure 7.23: Erasing an element from the list

Size Getting the size of a linked list is not as easy as an array. In fact to find out how many elements are connected together a complete traverse of list is necessary. So its convenient to accept the overhead and store the size of the list for increasing its performance.

Resize While a list does not have a maximum size or any restricted size, this process consist of adding new elements or erase some others to achieve the given size of the list.

Swap Swapping two lists is very simple and consist of swapping only the head pointer of the two lists. Figure 7.24 shows this procedure.

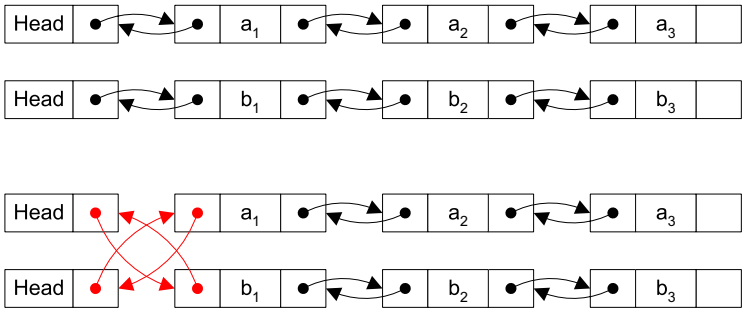


Figure 7.24: Swapping two list by changing their head pointer

Doubly Linked List Advantages

- Fast inserting a new element after or before a given one.
- Efficient removing of an element.
- A linked list can grow smoothly by adding new elements and without the need to resize it or creating buffer like for dynamic arrays.
- Removing elements from lists automatically reduces its size in memory which avoid the corresponding overhead or manual controlling of its size.
- Doubly linked list can be traversed in both way which makes it usable for a wider range of algorithms than the singly linked list.

Doubly Linked List Disadvantages

- Accessing by position is a very slow task and has to be avoided. This accessing can be reduced by keeping the pointer to the necessary elements of the list for future use.
- Lists are used to be less cache efficient due to the fact that its elements can be distributed arbitrary in the memory. This makes it less attractive as a fast iterating container in practice.
- The overhead of two pointers per element can be noticeable when elements are small. For example for a list of doubles these pointers can duplicate the memory in 32 bit compiling and even worse when compiling in 64 bit.

This container is suitable for storing data with high amount of insert and erase in the middle.

Implementation

C++ standard library provides a list class to represent the doubly linked list. This class is parameterized by template to accept different types.

```
list<class T, class Alloc = Allocator<T> >
```

7.2.5 Binary Tree

A binary tree is a tree in which every node has either no children, only a *left child*, only a *right child*, or both left child and right child. In the other words each node in a binary tree has two descending branches, left and right, which can be empty or not. Figure 7.25 shows a binary tree.

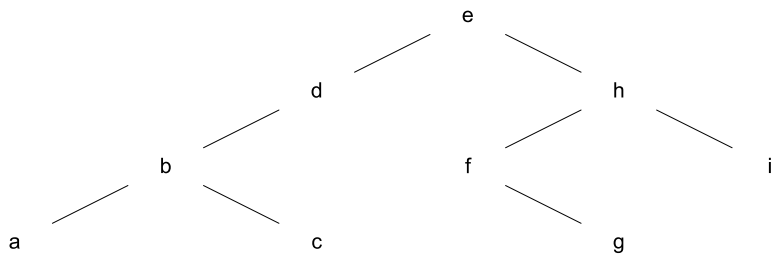


Figure 7.25: A binary tree

It is important to mention that the order of left and right child is part of the tree specification and swapping the left child and the right child of a node results in another binary tree deferent from the original one. For example trees in figure 7.26 are not equivalent due to the change of child positions.

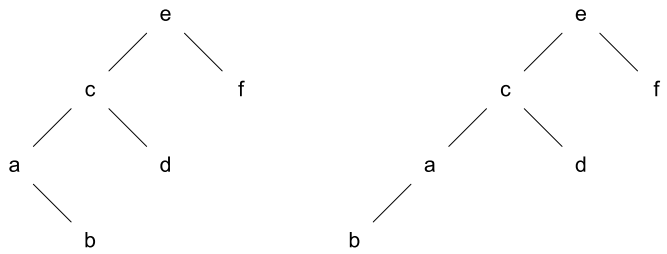


Figure 7.26: Two different binary trees because the position of node b is changed from being left child to be right child

Binary tree can be used for ordering data respect to a binary comparison operator which returns true or false. For example a less than operator $<$ can be used to order a set of numbers in a binary tree as can be seen in figure 7.27. In this tree each node is greater than all values in its left subtree and less or equal to all values in its right subtree. This ordering can be used later for searching a value in the tree.

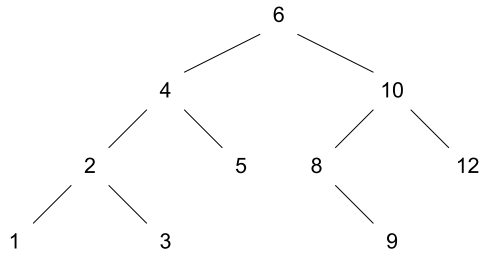


Figure 7.27: A binary tree containing a set of numbers ordered respect to the less than operator $<$

Interface and Operations

Access In a binary tree its not usual to give access by position and access is usually done by finding a key in the tree.

Insert Inserting a new element in a tree consists of finding its place and then add it there. In some implementations a given position is also accepted but as a hint to find faster the correct places. This searching is necessary to guarantee the ordering of the tree. The procedure of finding the right place to insert starts from the root and compares the new element key with the root. If the result is true goes to the left child and if it is false goes to the right child and repeat the comparison. This procedure is repeated until an empty child is founded which is the position to insert the element. Having this position inserting is only pointing the parent to this new element like inserting in linked lists. Figure 7.28 shows this process for inserting the value 7 in the ordered tree of figure 7.27.

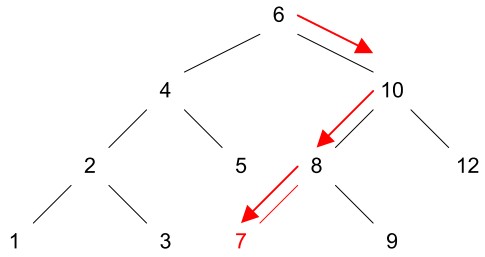


Figure 7.28: Inserting a new value in the tree.

Find To find a value in a tree one can began from root and compare the value with it. If comparison returns true means the value must be in left subtree and if it is false, means that the value must be in right subtree. Having the corresponding subtree the procedure can be repeated to see which subtree should has the value. This process will arrive either to our desirable variable or to an empty branch which indicates that the value does not exist. The ordering and hierarchial nature of the tree makes the finding algorithm very efficient. The number of comparisons required for this search is guaranteed to be $O(\ln N)$ which is much lower than the $O(N)$ comparison required for searching in an unordered normal array, specially for big number of entities N. Figure 7.29 shows this procedure in finding value 9 in the ordered tree

of figure 7.28.

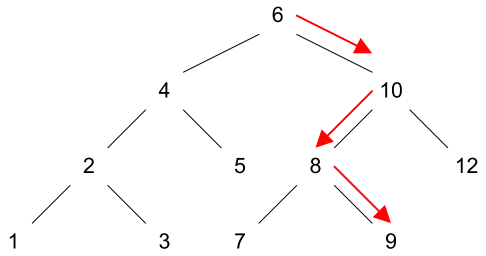


Figure 7.29: Finding a value in a binary tree

Erase Erasing an element of a binary tree requires some replacements of elements in order to keep the ordering of the tree. As mentioned earlier the ordering ensures that comparison of a node value with all the values in its left subtree is true and that with all its right subtree is false. In the case of using the less than operator $<$ for comparison it can be said that all values in left subtree are smaller than node value and all value in right subtree are greater than it. It can be seen that to keep this ordering we must take the minimum value, or the left most value of the right subtree and put it in the place of the removed node. For example removing 6 from the binary tree of figure 7.29 consists of first, finding the left most node of the right subtree which is 7 and put it in the place of 6. Then filling the empty place of the moved minimum node with its right child which is empty in this case. The resulting binary tree can be seen in figure 7.30.

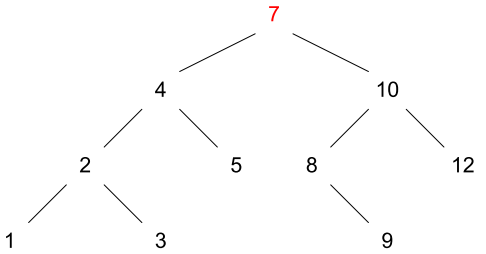


Figure 7.30: The binary tree after erasing an element

Iterating Unlike arrays and linked lists, iterating over an ordered binary tree is not so trivial and consist of ascending and descending the tree structure to keep track of the elements sequence. The idea is to take the start node and go into its right subtree, In each subtree the iterator finds the left most element and tries to go from left to right by going up and down in the tree layers. Figure 7.31 shows the iterator path from element 1 to element 12.

Size Like linked lists getting the size of a binary tree is not easy and needs a traverse over the tree. It is therefore convenient to accept the overhead and store the size of the tree in a member variable for increasing its performance.

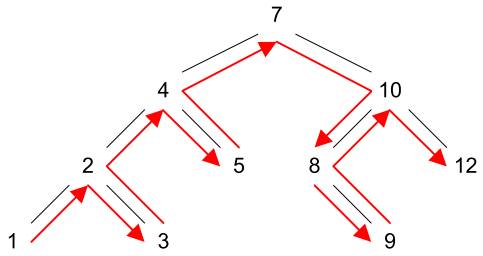


Figure 7.31: Iterator path from elements 1 to 12

Swap Swapping two binary trees is very simple and consist of swapping only the root pointer of two trees.

Binary Tree Advantages

- Keeping elements in order all the time is an important feature specially for working with variable set of elements.
- Fast finding procedure. The hierarchical structure and order nature of binary trees enable us to make a very efficient binary search also over very large number of elements.
- Relatively fast inserting a new element while keeping the ordering of elements.
- Relatively efficient in removing elements again without altering the ordering of elements.

Binary Tree Disadvantages

- There is no access by position in general. They are specialized for accessing by key and not by position.
- Binary lists are not cache efficient because their elements are not stored sequentially in memory.
- Iterating over a binary tree is a complex and consequently not efficient process.
- The overhead of three pointers per element can be noticeable when elements are small.

Implementation

Fortunately the C++ standard library provides various classes representing binary trees in different ways. The first one is `set` which is an ordered binary tree of its template argument type. It also takes an optional comparison operator which make it more generic. `set` uses this comparison to understand the order of an element.

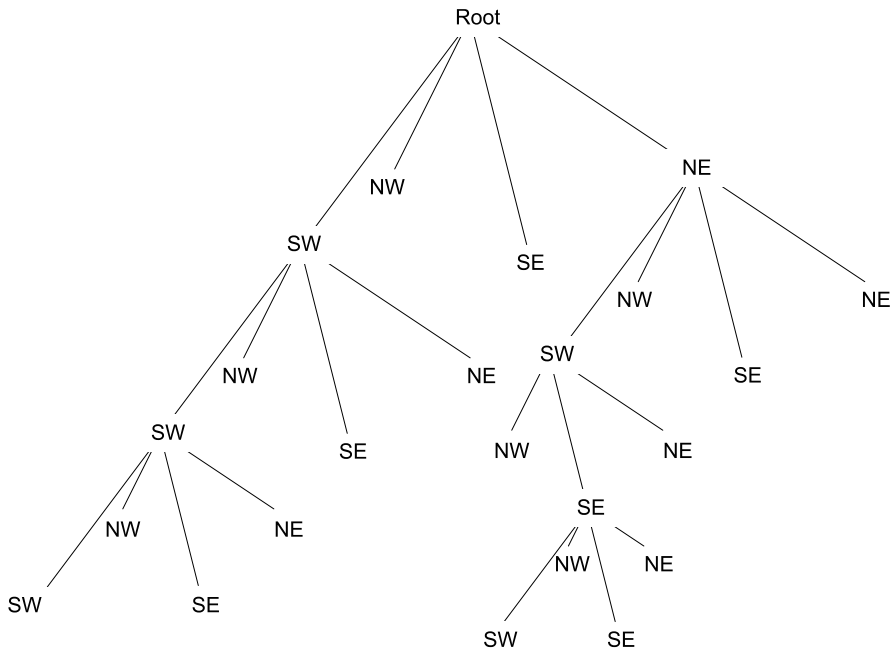


Figure 7.32: In a general quadtree each node has up to four children

7.2.6 Quadtree

Quadtree is a specific type of tree in which every node has zero to four children. Figure 7.32 shows a general quadtree.

This structure is very useful for organizing data in two dimensional spaces because it can be used to handle the partitioning information which defines the cells in space in hierarchial form. Figure 7.33 shows a domain divided by quadtree. For this reason the names of children in a quadtree node come from their relative positions in a two dimensional map: NW, NE, SW and SE.

As mentioned above, a binary tree can be used for organizing data in one dimension. Quadtree does the same in a two dimensional space. It can be used with two comparison operators to order points respecting their coordinates. For example using two less than operators $<$ results in an order quadtree in which the coordinate x of each node is greater than all coordinates x in its NW and SW subtrees and its y coordinate is greater than all coordinates y in SW and SE subtrees, as can be seen in figure 7.34.

Interface and Operations

Access In a quadtree accessing to an element is done by finding a pair keys, for example two coordinates of a point, in the tree.

Insert Inserting a new element in a quadtree consists of finding its place and then add it there. Sometimes a given position is accepted as a hint to find faster the correct position. The procedure is similar to binary tree but using two comparison to find the correct branch. It starts from root and goes through branches until a branch leads to an empty child. Having this position inserting is only pointing the parent to this new element.

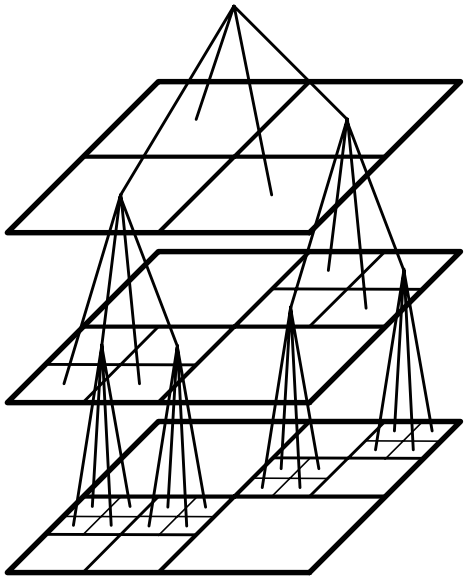


Figure 7.33: Quadtree can be used to partition a domain into layers of sub-domains.

Find Finding procedure starts from root and compares the keys, for example the point's coordinates, if matches to it the result is found and if not goes to the branch which is corresponding to the comparison result. The procedure is repeated for the node in the branch and keeps going until finding the entity or an empty leaf which indicates that the entity does not exist.

Different types of the quadtree and detail description of each type can be found in [91, 92]

7.2.7 Octree

Octree implements the same concept of quadtree but in a three dimensional space. In octree each node has 8 branches which may lead to a child or not. Considering the octree like the three dimensional extension of quadtree, all operations done by quadtree in two dimensions and using two comparison operators, now can be done by octree in three dimensions using three comparison operators.

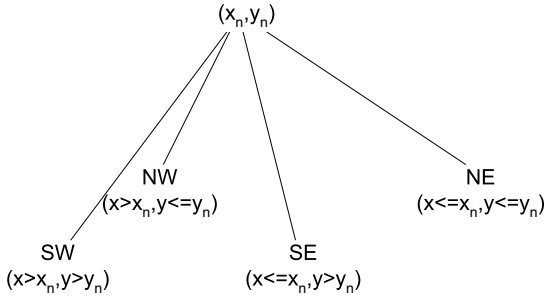


Figure 7.34: Ordering two dimensional points in quadtree.

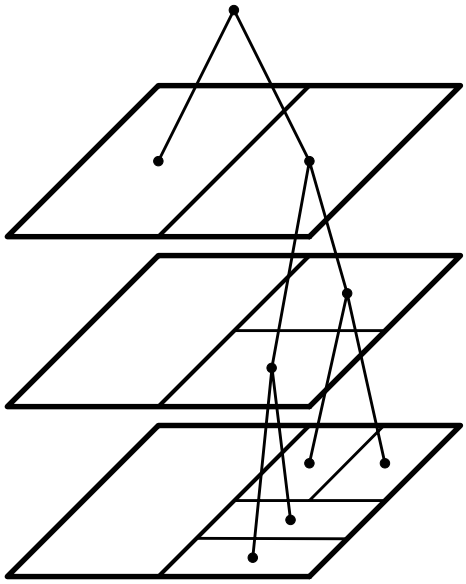


Figure 7.35: Partitioning a two dimensional domain using k-d tree.

7.2.8 k-d Tree

K-d tree is a generalized approach to deal with k dimensional spaces using only two-way branching at each node. Figure 7.35 shows a k-d tree used to handle the partitioning of a two dimensional domain.

7.2.9 Bins

A simple but effective data structure for storing and finding objects in two and three dimensional spaces is `Bins`. It divides the domain into a regular $n_x \times n_y \times n_z$ sub-domains and holds an array of buckets storing its elements. Figure 7.36 shows a two dimensional domain divided by the bins and figure 7.37 shows the structure of this bins.

This structure provides a fast spatial searching when entities are more or less uniformly distributed over the domain. The good performance for well distributed entities and simplicity make bins one of the popular data structure in different finite element applications [59].

7.2.10 Containers Performance Comparison

As mentioned before, each container respecting to its internal structure provides different performances in its operations and memory consuming. This difference has been described as advantage and disadvantage of each container. However this respective performance is also depends on the size of the container and can change radically in terms of the size. In this section a brief comparison between containers on certain operations is provided. This comparison gives some experimental results, indicating the behavior of different containers in practice.

It is important to mention that in these tests no manual optimization is performed and only the automatic optimization of compilers is set to its highest value.

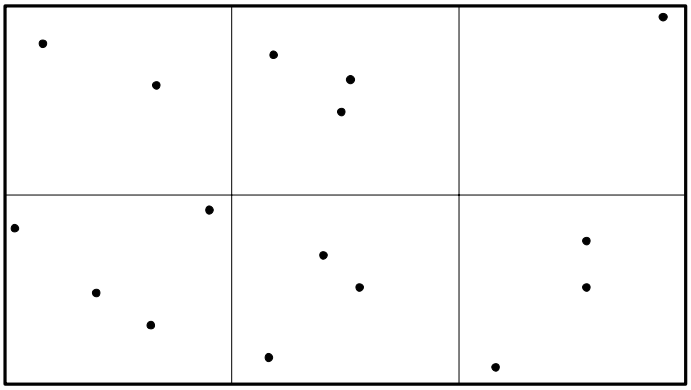


Figure 7.36: Partitioning a two dimensional domain using bins.

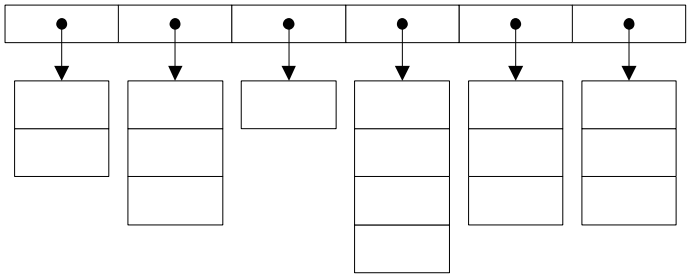


Figure 7.37: Bins structure.

Memory usage of containers

In this comparison different containers are initialized to different sizes and the memory used by each of them is measured. Figure 7.38 shows the results of this comparison.

It can be seen that `vector` uses less memory than other contatiners. The reason is the memory used to store the pointers in elements.

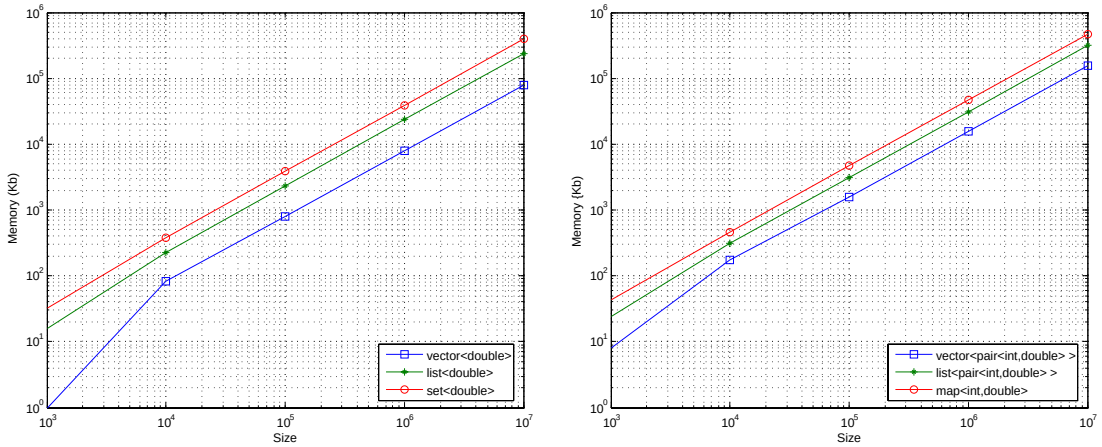
Another benchmark is done to see the memory used by different containers respecting to their sizes. For this reason an array of size $n = 100000$ is created with different containers and the memory used by each of them is shown in figure 7.39.

Construction and Destruction

In this test the construction and destruction time of different containers are compared. Figure 7.40 shows the construction time comparison for different containers.

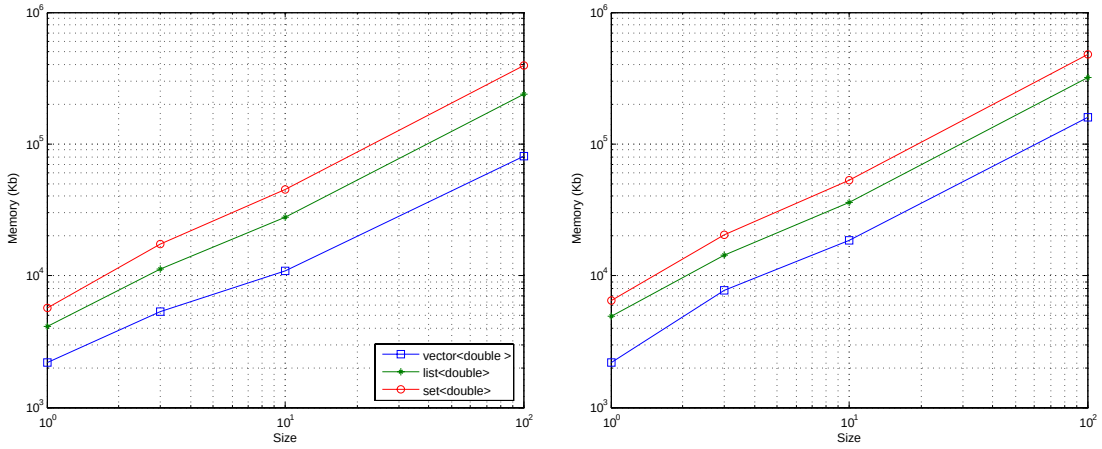
It can be seen that the constructing time for `vector` is far less that for other containers. The reason is its simpler internal structure than the `list` or `set`. Figure 7.41 shows the destruction time for different containers.

Again `vector` is far faster than other containers. This makes `vector` a good choice for situations



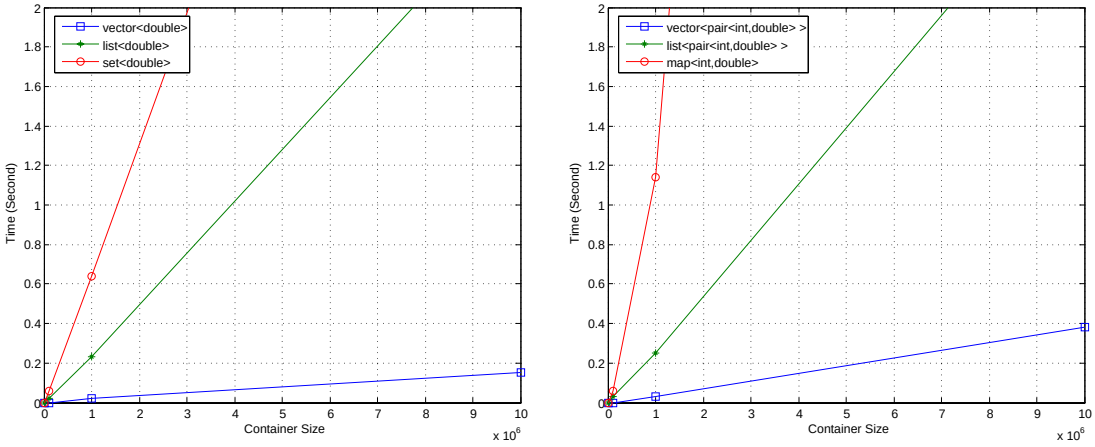
(a) (b)

Figure 7.38: Memory use comparison a) Comparing memory use of `vector<double>`, `list<double>` and `set<double>` b) Comparing memory use of `vector<pair<int,double> >`, `list<pair<int, double> >` and `map<int,double>`



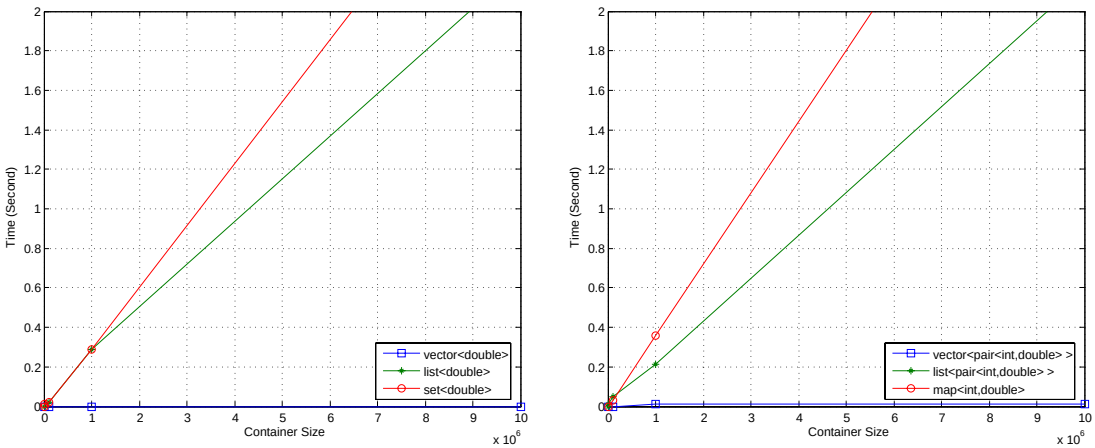
(a) (b)

Figure 7.39: Memory use comparison between arrays with size $n = 100000$ of different containers. a) Comparing memory use of `vector<double>`, `list<double>` and `set<double>` b) Comparing memory use of `vector<pair<int,double> >`, `list<pair<int, double> >` and `map<int,double>`



(a) (b)

Figure 7.40: Construction time for different containers. a) Comparing construction time for `vector<double>`, `list<double>` and `set<double>` b) Comparing construction time for `vector<pair<int,double> >`, `list<pair<int, double> >` and `map<int,double>`

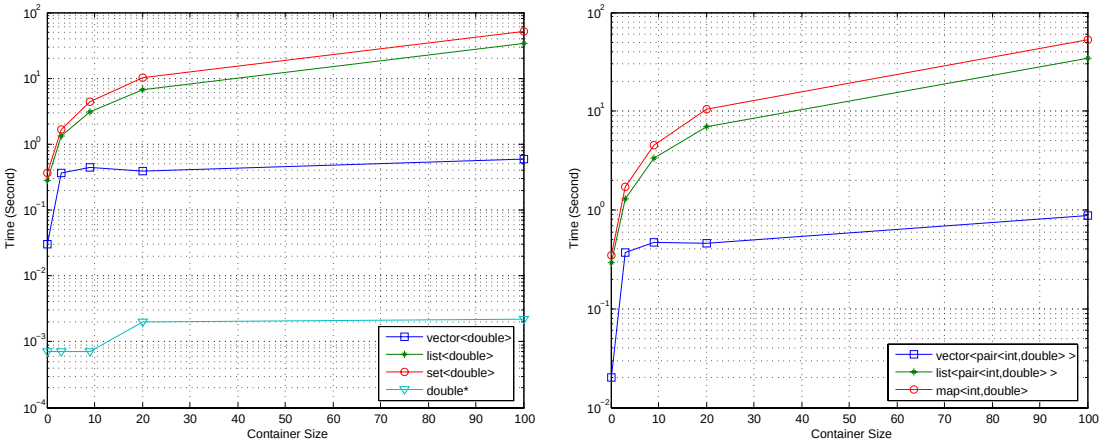


(a) (b)

Figure 7.41: Destruction time for different containers. a) Comparing construction time for `vector<double>`, `list<double>` and `set<double>` b) Comparing construction time for `vector<pair<int,double> >`, `list<pair<int, double> >` and `map<int,double>`

that a container has to be created and deleted immediately.

Another test is to compare the construction and destruction time of containers as local variables allocated in stack memory. While it is usual to create containers as local variables in procedures it is important to see their time overhead for construction and destruction them each time the procedure is called. Figure 7.42 shows this comparison.



(a) (b)

Figure 7.42: Construction and destruction time of containers defined as local variables and allocated in stack. a) Comparing construction/destruction time for `double*`, `vector<double>`, `list<double>` and `set<double>` b) Comparing construction/destruction time for `vector<pair<int,double> >`, `list<pair<int, double> >` and `map<int,double>`

This time the `C` static array shows far better performance even respecting to `vector`. The reason is the overhead of allocating dynamic memory for the `vector` while the `C` static array is allocated completely in stack. Consequently, implementing the small local containers in procedures as static arrays can significantly increase the performance of the code.

Iterating

Many finite element algorithms are used to iterating over containers. For this reason good performance in iterating is an important factor in selecting a container. This benchmark consists of iteration over all elements of sample containers with different sizes. The time is measured for 10^9 steps of iterations and the results are shown in figure 7.43. Each step consists of an access to the iterator content to be sure that the optimizer will not eliminate the loop.

This benchmark shows the bad performance of containers with tree structure like `set` and `map`. Surprisingly `vector` shows to be more robust in optimizing the iteration time than the `C` array and it seems that c array needs manual optimization to get its best performance.

Inserting

The first benchmark shows the results of pushing back elements to different containers. The test consist of pushing back 10^4 elements to each container several times and measuring the average

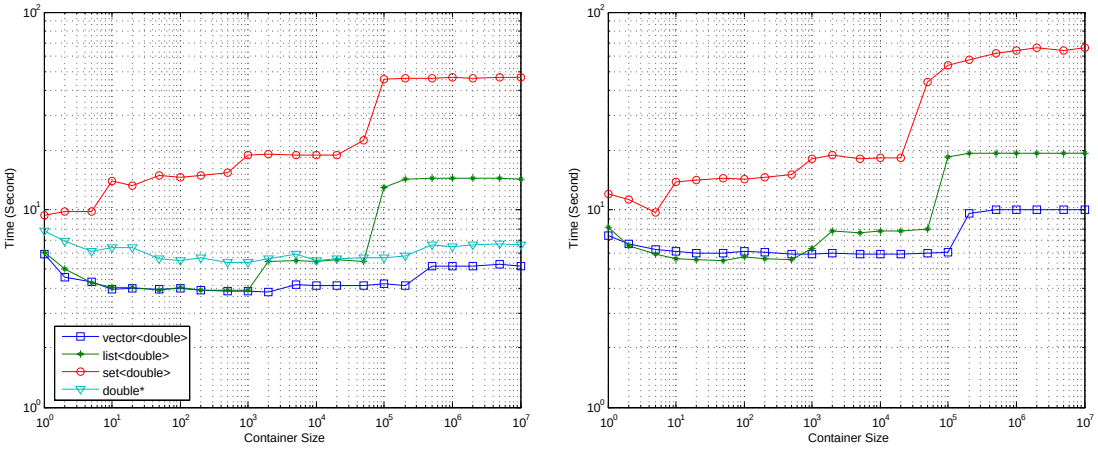


Figure 7.43: Iterating time for 10^9 steps of iterations with different containers. a) Comparing performance of `double*`, `vector<double>`, `list<double>` and `set<double>` b) Comparing performance of `vector<pair<int,double> >`, `list<pair<int, double> >` and `map<int,double>`

time of this operation. Figure 7.44 shows the results of this benchmark.

The second benchmark shows the push front operations for vector and its linear complexity with respect to the size of the container. Other containers has the same performance as the push back operation and are significantly faster as expected. Figure 7.45 shows the result of 10^4 pushfront to vectors with different sizes.

Copying

Another important operation is the copying of containers. This benchmark shows the result time for copying different containers with different sizes. The test consists of calling the copy constructor of container several hundred times and calculating the elapsed time for 100 times calling the copy constructor. Figure 7.46 shows the results of this benchmark.

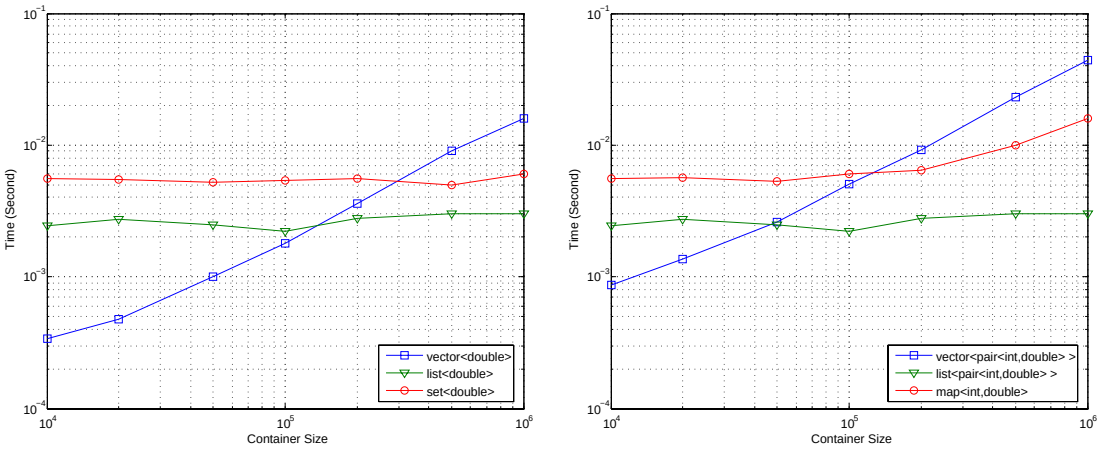
It can be seen that again `vector` has better performance to others due to its simple structure which leads to less overhead in time of copying than the others.

Find

Finding an element in a container is another typical operation to be compared. The benchmark compares the performance of the brute-force search over unsorted containers with binary search over sorted containers. The first test is to find a value in container, using brute-force over an unsorted `vector` and an unsorted `list`, a binary search over a sorted `vector` and the tree search of `set`. The second test is an integer key finding using brute-force over an unsorted `vector` and an unsorted `list` and the tree search of `map`. Figure 7.47 shows the results of this comparison.

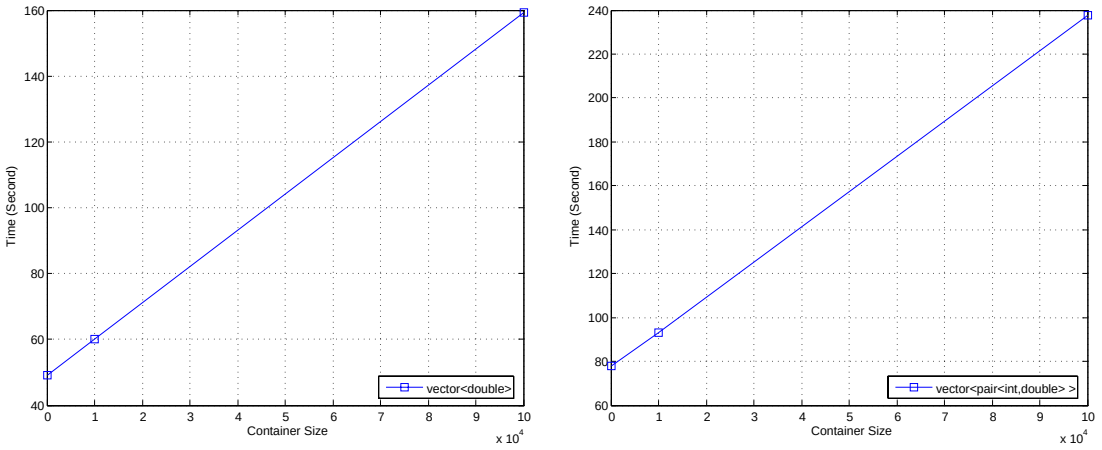
This benchmark shows how inefficient the brute-force algorithm becomes when the size of container increases. Also it can be seen that using a binary search over a sorted vector can lead to the same performance of searching in a tree like set.

Figure 7.48 shows the same results but focusing only on small containers. Here it can be seen



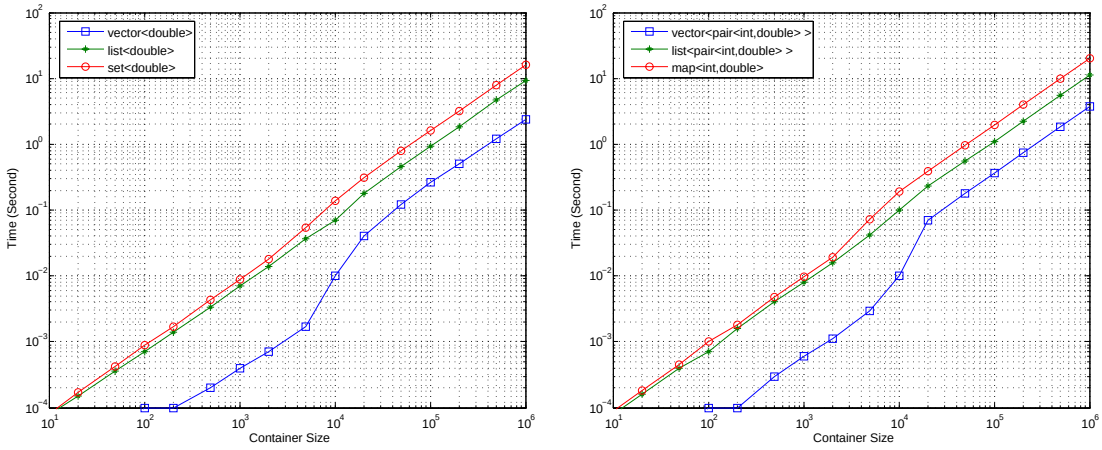
(a) (b)

Figure 7.44: Time comparison for 10^4 elements pushback to different containers. a) Comparing performance of `vector<double>`, `list<double>` and `set<double>` b) Comparing performance of `vector<pair<int,double> >`, `list<pair<int, double> >` and `map<int,double>`



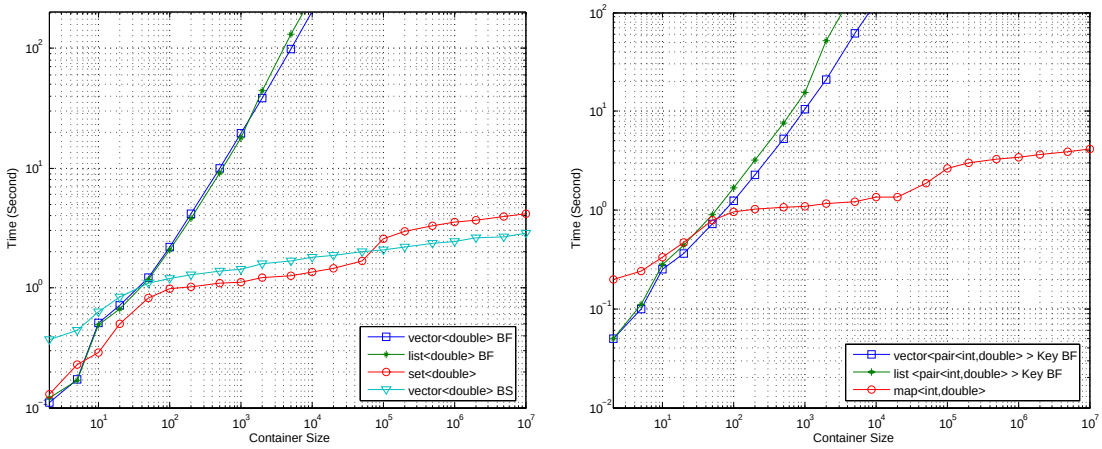
(a) (b)

Figure 7.45: Time comparison for 10^4 pushfronts to vectors with different sizes. a) `vector<double>` b) `vector<pair<int,double> >`



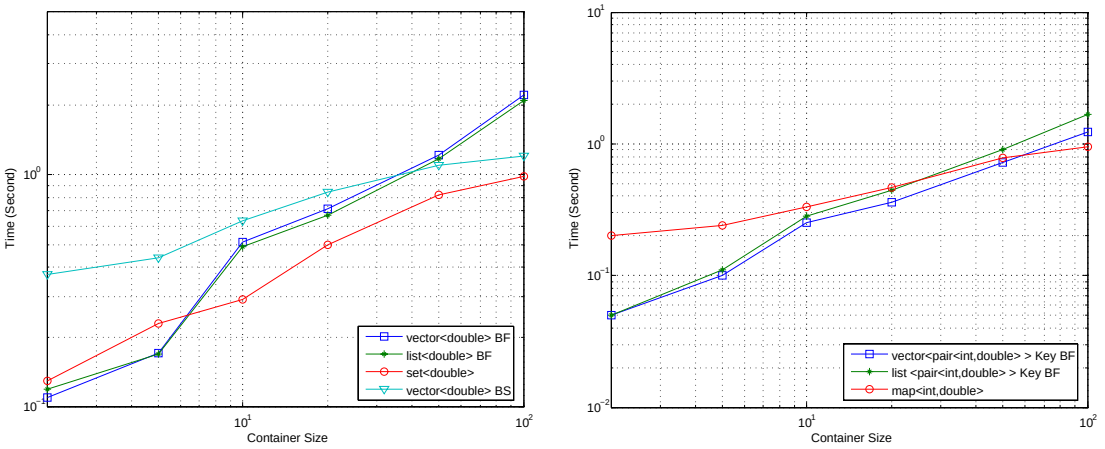
(a) (b)

Figure 7.46: Time comparison for 100 calls to copy constructor. a) Comparing performance of `vector<double>`, `list<double>` and `set<double>` b) Comparing performance of `vector<pair<int,double> >`, `list<pair<int, double> >` and `map<int,double>`



(a) (b)

Figure 7.47: Time comparison for different searching algorithms over sorted and unsorted containers a) Comparing performance of `vector<double>` with brute-force, `list<double>` with brute-force, `set<double>` binary tree search and sorted `vector<double>` with binary search. b) Comparing performance of `vector<pair<int,double> >` with brute-force key finding, `list<pair<int, double> >` with brute-force key finding and `map<int,double>` binary tree search.



(a) (b)

Figure 7.48: Time comparison for different searching algorithms over sorted and unsorted small containers a) Comparing performance of `vector<double>` with brute-force, `list<double>` with brute-force, `set<double>` binary tree search and sorted `vector<double>` with binary search. b) Comparing performance of `vector<pair<int,double> >` with brute-force key finding, `list<pair<int, double> >` with brute-force key finding and `map<int,double>` binary tree search.

that using a brute-force algorithm for small containers can lead to better performance than other algorithms. This feature will be used later in finding values in small data containers.

7.3 Designing New Containers

It can be seen that standard C++ containers are homogeneous due to the static typing of C++ language. This means that using one of this containers to store doubles cannot be reused to store vectors and matrices without decomposing them to double values. This restriction makes developers to separate their containers for different types of data or to implement heterogeneous containers. In this part some new containers capable to store different types of data are introduced.

In previous chapter a Variable Base Interface (VBI) was introduced and its advantages were mentioned. In this part VBI is used to unify the interfaces of different containers and also to provide them with a generic interface capable to storing any new variable without rewriting them.

7.3.1 Combining Containers

Finite element developers usually work with integers, reals, vectors and matrices as their data. In some other fields like electro magnetic formulations complex numbers are also used. So making a new container capable to hold just these data types can cover a large part of finite element programming needs. A very fast an easy way to implement a quasi heterogeneous container holding above data types is to take different containers and put them together as a new container. Figure 7.49 shows an example of this container.

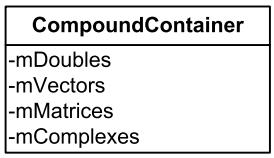


Figure 7.49: Combining containers for holding doubles, vectors, matrices and complex numbers.

Interface and Operations

Access For accessing to an element this container first has to see what is the type of this element and then access in corresponding container. Usually this access consists of a find process for a given key. However access by position can be done by giving a pseudo order to the containers.

Insert Inserting a new element like accessing an existing one requires switching over element type. The process is simply finding the corresponding container and insert the new element in it.

Find Mainly depends on the sub-containers inside the compound one. Again this container only dispatches the request to the corresponding sub-container using type of data and the procedure of finding must be done by the underlying container.

Erase Simply removes an element from the container holding this type of element. Efficiency of this procedure also depends on the type of container holding this data.

Iterating The same iterating mechanism of sub-container can be reused to iterate over a certain type of data.

Combining Containers Advantages

- Fast and easy implementation. Standard containers can be reused here and make the implementation task very easy.
- Very rigid and errorless structure. There is no chance in inserting wrong data type or getting data which is not of the expected type. Everything relies on C++ static type checking without dangerous type casting and raw pointer manipulation.
- Keeping separated different types of data lets more specialization for each case. For example in time of copying the containers for built in types can be copied directly in the memory and so on.
- Less searching time because the number of data in each container is less than the total number of data in container. In other words dividing container to sub-containers reduces the time for searching. Though the searching time is highly dependent on the type of sub-containers.

Combining Containers Disadvantages

- Extra memory overhead is needed for supporting any new type. As mentioned in previous section, each container has a memory overhead. So using more containers to keep the same number of element increases the overhead per element. This factor may cause problems

for very small number of data per container. For example a container for holding doubles, vectors, matrices and complex numbers has a fix overhead of 8 pointers or more. So using this container to hold a double and a vector of 3 doubles causes at least 100% memory overhead. Supporting any new data type still increases more this overhead which may cause this container unusable for some problems.

- Adding new types needs modifying the container, though this modification is very small. A good implementation can minimize this modification but cannot make it automatic to accept any new type of data. This makes it unacceptable for a library with unspecified using field.

Implementation

Combining containers is an easy task. Any classic container described in previous section can be use here to store one type of data. There is no restriction to have same containers for all types but usually this is the good choice while the nature of data and algorithms are not dependent on the type of element. However in some cases a type specific optimization can be implemented using different algorithms and containers for the data types.

As mentioned before a container provides interface for accessing, inserting, erasing and also iterating. Even though these are the common interfaces for standard containers, their type dependency given them a different nature. A simple way to overcome this problem is to implement separate methods for each type. Figure 7.50 shows an example of the accessing methods for the container of figure 7.49 using separate methods for each type.

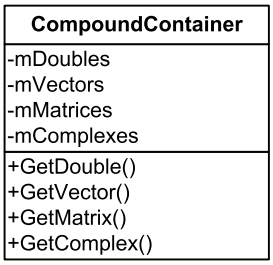


Figure 7.50: Implementing separate access interface for each type in a compound container

It can be seen that this interface design causes a big overhead in implementation cost. This can be avoided using a template access method with a dummy argument indicating the type of data. Here is an example of a template access method:

```
template<class TDataType>
  TDataType& GetValue(TDataType const& Dummy,
                      ...more data information)
  {
    return CorespoundingContainer.Get(...);
  }
```

Using this interface implies introducing a dummy variable just to help the container which is not elegant. In previous chapter the VBI was introduced and its generic way to deal with different data type was also discussed. Now it is time to use this concept to create a generic and extendible interface for our containers. VBI provides a uniform template model for accessing an element by

variable. This model uses the variable type parameter to distinguish the type of element without the need of dummy argument. In this manner the access method can be written in this new form:

```
template<TVariableType>
  typename TVariableType::Type& GetValue(TVariableType const&)
  {
    return CorespoundingContainer.Get(...);
  }
```

A variable not only gives information about the type of element but also gives information about how to find it by providing its unique index and also its name. This information can be used to find the element without passing extra arguments to the access method. So this access method can be written in the following way:

```
template<TVariableType>
  typename TVariableType::Type&
  GetValue(TVariableType const& ThisVariable)
  {
    return CorespoundingContainer.Get(ThisVariable.Key());
  }
```

This encapsulation of element information in a variable simplifies the interface and its usage by the users. Now a user can get a value only by giving the variable represent it as follows:

```
// Without VBI user must know exactly the type of
// data and also its index in container. Here is
// an example of accessing acceleration with index
// 3 in container
acceleration = mData.GetValue(array_1d<double, 3>(), 3);

// Using VBI, user just need to specify the previously
// defined variable
acceleration = mData.GetValue(ACCELERATION);
```

VBI also prevents users from making trivial errors by giving wrong type or information. For example:

```
// Error! acceleration type is array_1d<double, 3>
// but given type is std::vector
acceleration = mData.GetValue(std::vector(3),3);

// Error! displacement is in position 0 and not 3
displacement = mData.GetValue(array_1d<double, 3>(), 3);
```

Keeping this form a basic interface for our heterogeneous container can be implemented in the form below:

```
// Accessing to a value in container
template<TVariableType>
  typename TVariableType::Type&
  GetValue(TVariableType const& ThisVariable);

// Readonly access to the container
template<TVariableType>
  typename TVariableType::Type const&
```

```
  GetValue(TVariableType const& ThisVariable) const;

// Setting a value in container
template<TVariableType>
  void
  SetValue(TVariableType const& ThisVariable
           typename TVariableType::Type& Value);

// To see if the variable exist in container
template<TVariableType>
  bool
  Has(TVariableType const& ThisVariable);
```

This interface can be used in this form without problems but can be improved even more. In finite element applications there are many situations where there is a need for accessing a component of an array or matrix. For example to assign a value to a degree of freedom representing Y component of displacement, a direct access to the component value in data structure is easier than extracting the whole displacement vector and assign to it manually, as shown in the following code:

```
void UpdataDofValueInContainer(Dof const& rThisDof, double Value)
{
    if(Dof.Variable() == DISPLACEMENT_X)
        mData(DISPLACEMENT)[0] = Value;
    else if(Dof.Variable() == DISPLACEMENT_Y)
        mData(DISPLACEMENT)[1] = Value;
    else if(Dof.Variable() == DISPLACEMENT_Z)
        mData(DISPLACEMENT)[2] = Value;
}
```

Having direct access to components makes above example as easy as follows:

```
void UpdataDofValueInContainer(Dof const& rThisDof, double Value)
{
    mData(Dof.Variable()) = Value;
}
```

It can be seen that the first form with manual accessing to components is not even general and works only for displacement components while the second form can be used for any defined component without problem. As a consequence we will implement the direct component access for our container.

Interface must be modified to distinguish a variable and a component of variable. In VBI this is an easy task while two different classes, `Variable` and `VariableComponent`, represent them. So interface can distinguish a variable from a component just by type of given argument. Overloading each method to accept either a `Variable` or a `VariableComponent` separates the implementation for this two concepts. It is very important to mention here that this separation is in compilation time and does not causes any cost due to type checking or other type recognizing mechanism. The first part of interface is related to get a variable of given type and process it. The type of variable is specified by a template parameter, to keep the generality of the interface.

```
// Accessing to a variable in container
template<TDataType>
  TDataType&
  GetValue(Variable<TDataType> const& );
```

```
// Readonly access to a variable in container
template<TDataType>
  TDataType const&
  GetValue(Variable<TDataType> const& ) const;

// Setting a variable in container
template<TDataType>
  void
  SetValue(Variable<TDataType> const&,
           TDataType& );

// To see if the variable exist in container
template<TDataType>
  bool
  Has(Variable<TDataType> const& );
```

The second part of interface consists of the same methods overloaded to accept a `VariableComponent` instead of a normal `Variable`.

```
// Accessing to component of a variable in container
template<TAdaptorType>
  typename TAdaptorType::Type&
  GetValue(VariableComponent<TAdaptorType> const& )

// Readonly access to component of a variable in container
template<TAdaptorType>
  typename TAdaptorType::Type const&
  GetValue(VariableComponent<TAdaptorType> const& ) const

// Setting a component of a variable in container
template<TAdaptorType>
  void
  SetValue(VariableComponent<TAdaptorType> const& ,
           typename TAdaptorType::Type& );

// Ask if container has this component which usually is
// equivalent to see if the variable holding this component
// is exist.
template<TDataType>
  bool
  Has(Variable<TDataType> const& );
```

Next we will implement the container. One way is implementing the container with sub-containers as its attributes. Figure 7.51 shows an example of this container with three sub-containers.

It can be seen that for each supported type an overloaded version of the interface methods is needed to call manually the corresponding container. For example the `GetValue` method must be overloaded for each type to call the `GetValue` method of sub-container which contains this type of data as follows:

```
// Accessing to a double in container
double GetValue(Variable<double> const& rThisVariable)
{
```

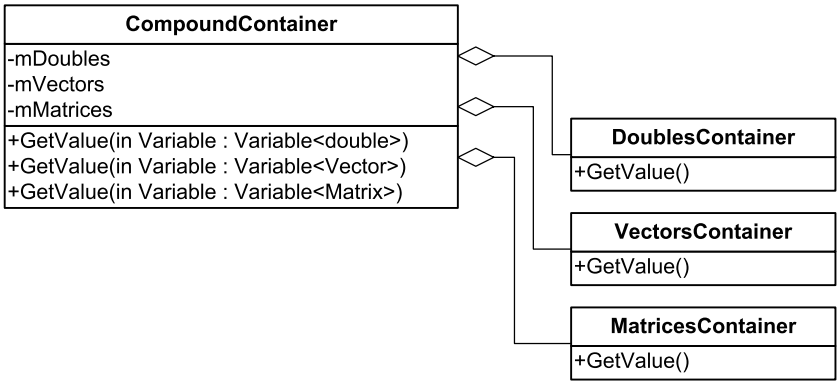


Figure 7.51: Container with three sub-containers as its attributes.

```
  return mDoubles[rThisVariable.Key()];
}

// Accessing to a Vector in container
Vector& GetValue(Variable<Vector> const& rThisVariable)
{
  return mVectors[rThisVariable.Key()];
}

// Accessing to a Matrix in container
Matrix& GetValue(Variable<Matrix> const& rThisVariable)
{
  return mMatrices[rThisVariable.Key()];
}
```

This manual switching must be done for all other methods working with different types of data like `GetValue const`, `SetValue`, `Has` and overloaded operators which makes this implementation strategy difficult to maintain.

Another approach is using multiple hierarchy to group different containers in a combine one. Figure 7.52 shows this approach for implementing the container of the previous example.

The big difference between this approach and the previous one is the container switching mechanism. In this way there is no need to manually overload each method for any acceptable type. Now a template method simply does the work. The mechanism is simple, each method is implemented as a template of the element type which accepts the corresponding variable and calls the related container automatically by calling the proper base class interface. Here is an example of this implementation:

```
class CombinedContainer : public BaseContainer<double>,
                                 BaseContainer<Vector>,
                                 BaseContainer<Matrix>
{
  public:

  /// Default constructor.
  CombinedContainer(){}
```

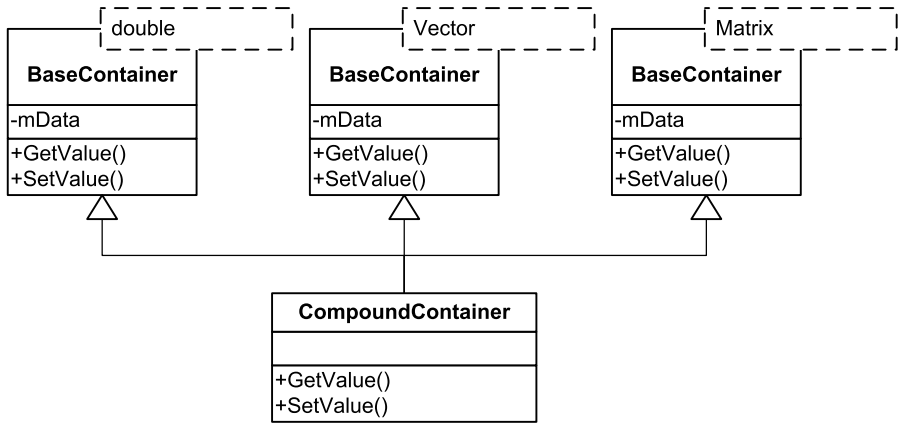


Figure 7.52: Combining different containers using multiple hierarchy.

```
  /// Copy constructor.
  CombinedContainer(const CombinedContainer& rOther) :
    BaseContainer<double>(rOther),
    BaseContainer<Vector<double> >(rOther),
    BaseContainer<Matrix<double> >(rOther)
  {
  }

  template<class TDataType>
  TDataType&
  GetValue(const Variable<TDataType>& rThisVariable)
  {
    return BaseContainer<TDataType>::GetValue(rThisVariable);
  }
};
```

Now supporting any new variables only needs another parent class to be added and modification in some other methods like copy constructor to incorporate the new base class.

Direct access to the components of variables can be done easily using tools provided by VBI. As mention in the previous chapter each component knows about its parent variable and also knows how to extract itself from it. So to extract a component is only necessary to access the parent variable value and give it to the component to extract itself form it. Here is an example of the component access method:

```
template<class TAdaptorType>
  typename TAdaptorType::Type&
  GetValue(const VariableComponent<TAdaptorType>& rThisVariable)
    {
      typedef typename TAdaptorType::SourceType source_type;

      return rThisVariable.GetValue(
                BaseContainer<source_type>::GetValue(
                        rThisVariable.GetSourceVariable()));
    }
```

It can be seen that all switching and accessing algorithms are implemented via templates, which make them very efficient. In time of compilation all variables and components types are known so the compiler can get the conversion algorithm provided by component and inline it inside the access code to eliminate the function call overhead. Finally optimizer will reduce all this code to the direct access method provided by adaptor which is equivalent to the hand written code.

Finding stored variables by their keys require searching in container. `map` provides a searching mechanism by given keys and can be used to find any variable key without any implementation cost. The `BaseContainer` can be implemented using `map` as follows:

```
template <class TDataType>
 class BaseContainer
 {
 public:
  typedef map<VariableData::KeyType, TDataType> ContainerType;

  TDataType&
  GetValue(const Variable<TDataType>& rThisVariable)
  {
    return mData[rThisVariable];
  }

 private:
  ContainerType mData;
 }
```

So a very first version of a container with VBI can be made by putting all above components together. This implementation was used in the first version of the Kratos code to develop a reliable container with minimum cost.

This container can be improved by changing the basic container from `map` to a `vector`. There are two major advantages in using `vector` instead of `map`:

- Less memory is needed to store a `vector` of indexed data than a `map`. In fact this container will be used to store nodal or elemental data, so any reduction in memory will deeply affect the overall memory used by the application. In the previous section the big difference in use of memory has been shown. For small containers `map` needs about 2 to 5 times more memory than `vector`. This overhead can be eliminated completely using vector.
- The `vector` searching time is also faster than `map` using even brute-force when the size of container is very small. Again respect to the fact that the amount of data to be stored for each `Node` or `Element` is small, so changing to vector can also make searching process faster.

The implementation is relatively easy. Each data will be stored in a `pair` combined with its key. So to find any data one must iterate over vector and compare the key with given one. A simple container called `VectorMap` can help us to encapsulate all these operations and reuse previous code by changing the standard `map` with it.

7.3.2 Data Value Container

A quasi heterogeneous container can be used successfully to hold data with small variety in their types. But to hold more different data types a heterogeneous container is more useful. Data value container is a heterogeneous container with a variable base interface designed to hold the value for any type of variable.

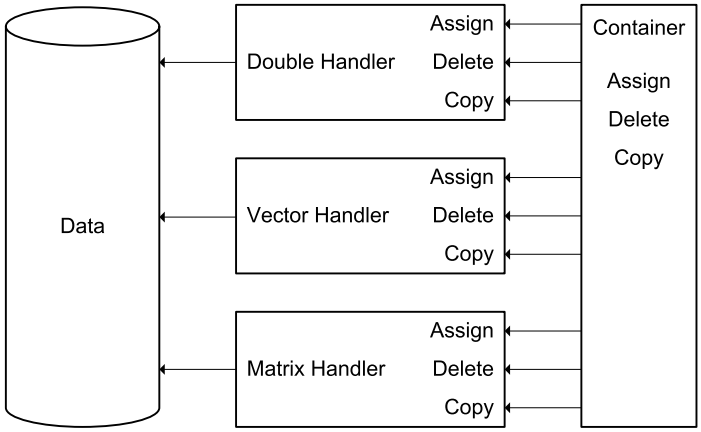


Figure 7.53: Heterogenous container uses different handlers to access data.

Usually a container needs to do some basic operations over its data. Creating, copying and deleting are examples of this operations. Unfortunately this operations may vary from one type to other. For example removing a double can be done by just removing it form container but removing a vector may consists of first freeing its memory and then removing it from container. So a heterogenous container needs a mechanism to handle each different type with its corresponding process. For example copy a double by copying its value and a vector by calling its copy constructor.

A common way to deal with this problem is to encapsulate all necessary operation into a handler object and associate it to its corresponding data. In this way the container only uses this handlers by their unique interface to do different tasks without any problem. Figure 7.53 shows this relation.

In our approach the variables are used as the handlers to help the container in its data operations. The data value container uses the `Variable` class not only to understand the type of data but also to operate over it via its raw pointer methods. Figure 7.54 shows the relationship of container and the `Variable` class.

Interface and Operations

Access Container first uses the key of given variable and finds the location of its value in memory. Then return this position as a reference with the type of variable. It is important that the type recognition can be done in compilation time in order to eliminate its overhead in runtime.

Insert Inserting a new element consist of allocating memory and copying correctly the object. Finding the correct position and allocating strategy depends highly on the internal implementation of data. By knowing the type of data, copying can be done easily by calling its copy constructor.

Find Mainly depends on the internal structure of the container and consists of searching the key of variable in the container. Unlike the compound container, the finding process does not need the type switching and the type of data is important only in time of down casting of returned value.

Erase Removing data consists of two parts. First calling the destructor of the object using the

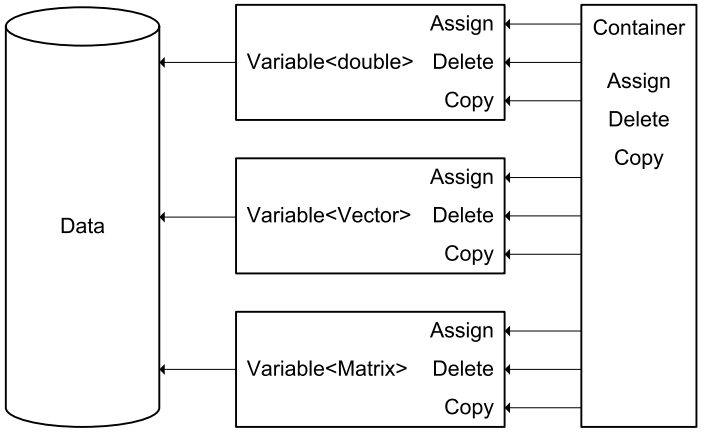


Figure 7.54: Data value container uses the `Variable` class to process its data.

type specification of the variable and then freeing the memory holding the value from internal data. The first part is an easy task as the type of variable is known at compilation time. The efficiency of the second part depends on the container's internal structure of data.

Copy Copying of this container is not a trivial task, as copying the memory block, or shallow copying, is not sufficient for copying some type of data. For example objects containing a pointer may need a deep copy of the pointed data and not the pointer itself. So the container goes element by element and uses the corresponding variable to copy the data. This procedure makes the process to be slower than for a homogeneous container.

Clear Clearing consists of first calling the destructor of each object and then freeing the memory. Container uses the variable to call the correct destructor of data and correctly remove it from the memory. This procedure is necessary because simply freeing the memory will not call the destructor of an object by itself. This causes problems specially when objects have some memory allocated internally and removing them without calling to destructor results in memory leaks in the system. Again this process is slower than for a homogeneous clearing due to the function calling overhead for each element.

Data Value Container Advantages

- Extendibility to store any type of data without any implementation cost. Adding new types to data value containers is an automated task without changing the container or even reconfigure it. One can store virtually any type of data, from simple data like an integer to a complex one like a dynamic array of pointers to neighbor elements.
- Usually extensive use of void pointers and down casting make heterogeneous container open to type crashing. Using the variable base interface protects users from unwanted type conversion and guarantees the type-safety of this container.

Data Value Container Disadvantages

- Heterogeneous containers are typically slower than homogeneous ones at least in some of their operations. The type recognition make them slower than for a homogeneous container.

Implementation

The first step to implement this container is designing the structure of data in memory. One approach is to group each data with a reference to its variable and put them in a dynamic array as shown in figure 7.55.

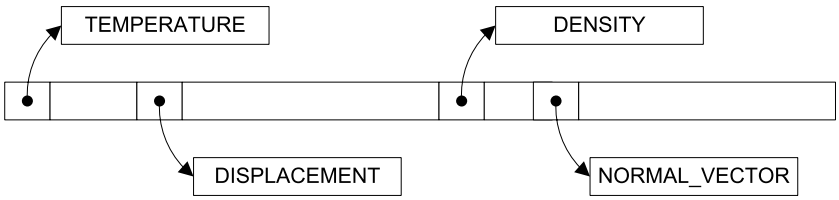


Figure 7.55: A data value container with continuous memory.

In this approach iterating over elements is somehow like a link list. For each element the variable knows the size of it and therefore the offset necessary for going to the next element. Having a pointer to each variable and not copying it is necessary to eliminate the unnecessary overhead of duplicated variables.

Another approach is to allocate each data separately and keep the pointer to its location in the container. Figure 7.56 shows a container with this structure in memory.

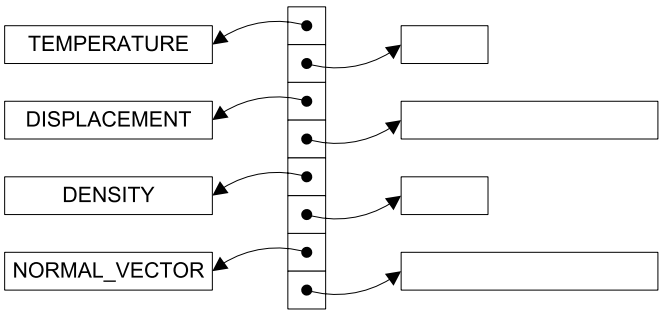


Figure 7.56: A data value container with discontinuous memory.

The first approach is typically more efficient in use of cache because accessing an element does not need a memory jump by a pointer. But adding new data to it may invalidate all references to its elements, as mentioned before for dynamic arrays. The second approach lets user to get a reference of its data once and use it several time without worrying about its validness. In this work the second approach is used because of its advantage in reducing the repeated accesses to the container which can increase significantly its overall performance in practice.

Using this memory structure the implementation is relatively easy. First a pair object is used to group the variable reference with a void pointer to its data:

```
// Grouping Variable reference with a pointer to its data
typedef std::pair<const VariableData*, void*> ValueType;
```

Now the internal data container can be implemented easily by putting these pairs in a vector:

```
// Type of the container used for variables
typedef std::vector<ValueType> ContainerType;
```

For inserting a new data in this container, first a new pair must be created which holds a reference to its variable and a pointer to the memory location holding the copy of the value. Adding this pair to the end of the vector finishes the inserting process:

```
mData.push_back(ValueType(&rThisVariable,new TDataType(rValue)));
```

Access methods use `Variable` or `VariableComponent` as data information as described for VBI. Each access consist of a find process for given variable key and then convert the data to the given variable type.

```
template<class TDataType>
  const TDataType&
  GetValue(const Variable<TDataType>& rThisVariable) const
{
  typename ContainerType::const_iterator i;

  i = std::find_if(mData.begin(),
                   mData.end(),
                   IndexCheck(rThisVariable.Key()))

  if (i != mData.end())
      return *static_cast<const TDataType*>(i->second);
}
```

The access method can be also configured to return zero if the given data does not exist yet in the container as follows:

```
template<class TDataType>
  const TDataType&
  GetValue(const Variable<TDataType>& rThisVariable) const
{
  typename ContainerType::const_iterator i;

  i = std::find_if(mData.begin(),
                   mData.end(),
                   IndexCheck(rThisVariable.Key()))

  if (i != mData.end())
      return *static_cast<const TDataType*>(i->second);

  return rThisVariable.Zero();
}
```

Its important to mention here that using VBI not only increases the readability of the code but also protects users from unwanted type crashing. To see the difference of these two approaches let us make an example of an access method without using VBI:

```
void*  GetValue(KeyType Key) const
{
  typename ContainerType::const_iterator i;
```

```
  i = std::find_if(mData.begin(), mData.end(), IndexCheck(Key))

  if (i != mData.end())
      return i->second;

  return &(rThisVariable.Zero());
}
```

Also considering the IO part of the application reads some data with different types from input and store them in the container as follows:

```
// Defining keys
  int acceleration_key = 0;
  int elasticity_key = 1;
  int conductivity_key = 2;

// Reading data
  array_1d<double> acceleration;
  matrix<double> conductivity;
  symmetric_matrix<double> elasticity;

  input >> acceleration >> elasticity >> conductivity;

// And store them in data value container
  data.SetValue(acceleration_key, acceleration);
  data.SetValue(conductivity_key, conductivity);
  data.SetValue(elasticity_key, elasticity);
```

In some other part of the code this variables are necessary and user will retrieve them from the container without specifying their types or with different types:

```
// The following innocent code simply will not compile!
// Error: There is no * operator which takes a double
//        and a void as its arguments
  Vector v = delta_time *  *data.GetValue(acceleration_key) + v0;

// The following erroneous code compiles without
// problem but crashes in runtime due to the type
// crashing of converting conductivity matrix to a
// double representing the conductivity coefficient!
  double k = *(double*)data.GetValue(conductivity_key);

// Again the following code will compile fine but crashes
// mysteriously in run time! Because the elasticity
// was stored as a symmetric matrix
  Matrix* d = (Matrix*)data.GetValue(elasticity_key);
```

The first statement calculates the velocity and store it as `Vector v`. This statement seems to be errorless, however simply will not compile because compiler has not any information about the type of acceleration.

The second statement is worse because it compiles without any problem but will not work as expected. So the application gives wrong results due to this type crashing and user has to debug it in order to find this simple error.

The third and more erroneous statement of taking a `symmetric_matrix` pointer as a pointer to

Matrix also will be compiled and even worse than second statement, may also work mysteriously or crash unfaithfully depending on the order of internal data in Matrix.

Now let see how using VBI protects user from simple errors by compilation time type checking:

```
// The following code works as it must while the return
// type of GetValue method is a reference to acceleration
// array. So the multiplication can be done correctly.
  Vector v = delta_time * data.GetValue(ACCELERATION) + v0;

// The following code will not compile due to the type
// mismatch.
  Matrix& d = data.GetValue(ELASTICITY);

// Again the following code will not compile due to the
// type mismatch.
  double k = data.GetValue(CONDUCTIVITY);
```

The first statement compiles without problem and also works as expected. A reference to the acceleration array is passed to the expression and the velocity vector will be calculated correctly.

Unlike the previous approach, the second statement will not compile because the compiler cannot convert a reference of symmetric_matrix type to a reference to Matrix type. This error in compiling protects users from unwanted type crashing. Also there is a possibility for users to copy correctly the elasticity symmetric matrix into a normal matrix for some subsequent operations.

Finally the third statement causes another compiling error and protects the user from erroneous type conversion.

Copying the container as mentioned before cannot be done by just copying the memory as this shallow copy results in the uncorrect copy of some objects, specially for ones with pointers to their individual data. For example let us consider a dynamic vector with the following implementation:

```
class Vector
{
    int mSize;
    double* mData;
public:
    // copy constructor
    Vector(Vector& Other)
    {
        mData = new double[Other.mSize];
        memcpy(mData,Other.mData,mSize);
    }

    // access
    double operator[](int i)
    {
        return mData[i];
    }
}
```

Shallow copying of this vector will result in the mData pointer taking the address in mData of the source vector and pointing to the source data as can be seen in figure 7.57. So any change in the elements of vector vc will change the elements of the source vector v!

But copying the same vector using its copy constructor will duplicate the allocated memory and safely uses the memcpy to copy the contents of the source vector to the copy one, as shown in

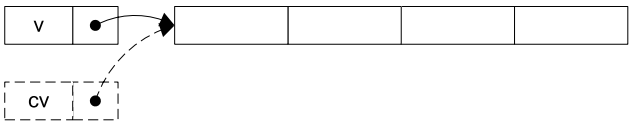


Figure 7.57: Shallow copying a pointer results in shared data for source and copied vectors.

figure 7.58.

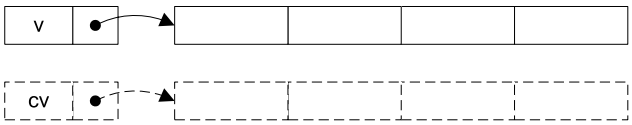


Figure 7.58: Deep copying results in an individual copy of source vector.

Data value container uses variables to call the copy constructor for each element at copying time in order to avoid errors produced by shallow copying. Here is the implementation for the copy constructor.

```
/// Copy constructor.
DataValueContainer(DataValueContainer const& rOther)
{
    for(ConstantIteratorType i = rOther.mData.begin() ;
        i != rOther.mData.end() ; ++i)
        mData.push_back(ValueType(i->first, i->first->Clone(i->second)));
}
```

Unfortunately the `Clone` method of the `Variable` class must be virtual and its function call overhead makes this operation slower than normal copying.

Destructing a data value container also need to be done carefully because freeing its memory can results memory leak for objects with internally allocated memory or result in unfinished jobs for some other objects. To avoid all these problems it is necessary to call the destructor of objects before removing them from memory. Data value container uses Delete method of Variable to call the destructor of each object in order to remove them correctly.

```
void Clear()
{
    for(ContainerType::iterator i = mData.begin() ;
        i != mData.end() ; i++)
        i->first->Delete(i->second);

      mData.clear();
}
```

This operation also consists of a function call in its loop which reduces its efficiency.

7.3.3 Variables List Container

In finite element programs it is common to store same set of data for all `Node`s of one domain. For example in a fluid domain each `Node` has to store velocity and pressure. Also each type of `Element` has to store an specific set of historical data at each integration point. The previous heterogeneous container can be used to store these data due to its flexibility to store any type of data. However the searching procedure in order to access data in this container makes it inefficient. To solve this problem another container is designed which stores only a specific set of data but with an efficient access mechanism.

The main idea is to use an *indirection* mechanism to access the elements of the container. A shared variable list gives the position of each variable in the containers sharing it. The mechanism is very simple. There is an array which stores the local offset for each variable in the container and assigning the value -1 for the rest of the variables. Offsets are stored in the position of variables key using a zero base indexing. In other words if the key of a variable is k, then its offset is stored as the $k+1$'th element of this array. This offset can be used to access the data in memory by offsetting the data pointer. For example to find temperature in this container, the key of the `TEMPERATURE` variable, in this example 2, indicates that the third element of the offset array contains the offset for temperature which is 1. Then this offset is used to get the value of temperature in the data array. Figure 7.59 shows this procedure.

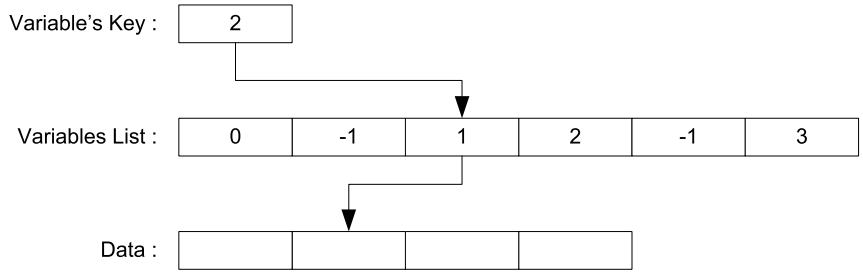


Figure 7.59: Accessing to a value in the variables list container.

Interface and Operations

Access This container first uses the key of a given variable and gets the necessary offset from the variables list. This offset is used to access its value in memory. Then return this position as a reference with the type of variable. Its important that the type recognition can be done in compilation time in order to eliminate its overhead in runtime. If the inserting in container is enabled, a control is necessary to see if the requested variable is really stored in the container or not.

Insert Inserting a new element consist of allocating memory and copying correctly the object and also adding it to the variables list. This means that adding a new variable to a container virtually adds it to all the other containers sharing the variable list with it. Using an array for data in this container implies that all references to elements of any container sharing the variables list can be invalidated by inserting a new element.

Find Finding is done via indirection and is a fast procedure. A variable key is enough to find it efficiently and only the type of data is needed for correctly casting the results of the search.

Erase Removing data is very complicated. It can be done by giving a removed tag to the variable in list and then each container has to update itself for new list. All these makes erasing practically unacceptable.

Copy Copying this container is similar to the `DataValueContainer`. Again the container goes element by element and uses the corresponding variable to copy the data. This process needs a virtual function call which reduces its performance.

Clear Clearing this container is also similar to the `DataValueContainer`. The container uses the variable to call the correct destructor of data and correctly remove it from memory. This procedure is necessary because simply freeing the memory will not call the destructor of object by itself. This causes problems specially when objects have some memory allocated internally and removing them without calling to destructor results in memory leaks in the system. Obviously this process is slower than an homogeneous clearing due to the function calling overhead for each element.

Variables List Container Advantages

- The accessing and finding processes are very efficient because only two indexing are necessary to find each value.
- Extendibility to store any type of data without any implementation cost. Adding new types to this container is an automated task without changing the container or even reconfigure it.
- Usually extensive use of void pointers and down casting make heterogeneous containers vulnerable to type crashing. Using the variable base interface protects users from unwanted type conversion and guarantees the type-safety of this container.

Variables List Container Disadvantages

- Having a shared variable list imposes an extra effort to group related containers and manage them in different groups. Practically this makes the object using this container less independent.
- Erasing a variable from this container is a complex and difficult task. For this reason variables list container cannot be used in problems with temporal variables.

Implementation

The first approach to implement this container is to put everything in the container and taking a simple list of variables to work. This approach looks attractive by encapsulating everything in the container and using an standard vector for the variable list. Unfortunately this design requires recalculation of the offset for each access which imposes an unacceptable overhead. So let us change the design to remove this unnecessary overhead.

Another approach is to divide the mechanism in two parts. One part for calculating the position and another for handling the memory. In this design the `VariablesList` class keeps the list of variables to be stored and also provides their local position by giving the necessary offset for each one. The container is in charge of allocating memory, copying itself and clearing the data in a correct way using a variables list. Figure 7.60 shows this structure.

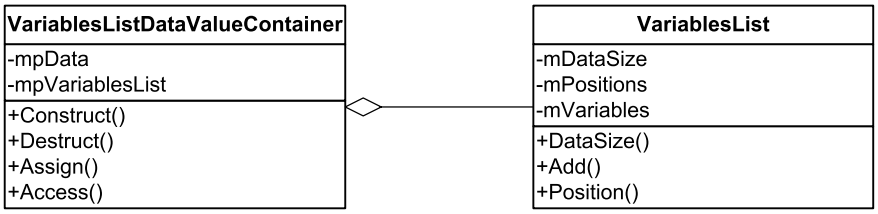


Figure 7.60: The `VariablesList` class provides the list of variables and their local positions for the `VariablesListDataValueContainer`.

An important decision here is to let container add new variables to the shared list or not?. How this feature affects our implementation? When each container is enable to add a new variable to the list of stored variables, this list also changes for all other containers sharing it. This change implies that for each access to a container, it must check if the list is changed or not and, in the case of new variables, update itself. This procedure introduces an overhead in all accesses to the container's elements. Also this updating invalidates all references to its elements which complicates more its use and increases the number of accesses to its data.

In Kratos the inserting was enabled to make this container compatible with previous ones. In practice the problem was not only the check and updating overhead. The updating makes debugging a difficult task. References are not reliable because there is no guarantee that some other container has not changed the list. This can be even worse in case of parallel computation because this change can happen in another thread just after a reference is taken. So finally the inserting feature was removed from this container to reduce problems using it.

Without the inserting ability the implementation of this container is very easy. Constructing is done by allocating the memory with data size provided by a given variables list. `VariablesList` is in charge of calculating the required memory to store its variables. To improve the portability of the code, the memory is divided into some blocks with a configurable specific size. `VariablesList` determines the number of blocks needed to store each variable and calculates the total size by the sum of the required blocks of memory. An additional step in the constructing process is assigning an initial value to elements in order to avoid uninitialized value problem. This can be done by using the `AssignZero` method of given variable which assigns its zero value to the allocated element in the container. The following list shows a default constructor for this type of container:

```
/// Default constructor.
VariablesListDataValueContainer()
  : mpData(0), mpVariablesList(&msVariablesList)
  {
    int size = mpVariablesList->DataSize()*sizeof(BlockType);

    // Allocating data using size provided by variables list.
    mpData = (BlockType*)malloc(size);

    // Initializing elements with zero value given by each
    // variable.
    VariablesList::const_iterator i_variable;

    for(i_variable = mpVariablesList->begin() ;
        i_variable != mpVariablesList->end() ;
```

```
        i_variable++)
        {
            std::size_t offset = mpVariablesList->Index(*i_variable);
            i_variable->AssignZero(mpData + offset);
        }
  }
```

The copying process for this container depends on its source. If the source is empty this just implies the clearing of the container. If it is not empty but sharing the same variables list, there is no need to perform the deallocation and allocation procedure of elements and the process consists of just assigning the values using the variables in the variables list. Finally for a source with different variables list it is necessary to clear the container and reallocating the memory for the new elements. The following code shows the assignment operator:

```
/// Assignment operator.
VariablesListDataValueContainer&
  operator=(const VariablesListDataValueContainer& rOther)
  {
    // if the source container is empty call clear.
    if(rOther.mpVariablesList == 0)
        Clear();
    // if other container uses the same variables list
    // assigns the container element by element.
    else if(mpVariablesList == rOther.mpVariablesList)
    {
        // Assigns other elements value using variable's assign
        // method.
        AssignElements(rOther);
    }
    else
    {
        // Destruct previous elements by calling their
        // destructors
        DestructElements();

        // Updates variables list
        mpVariablesList = rOther.mpVariablesList;

        // Reallocating the memory for new size
        int size = mpVariablesList->DataSize()*sizeof(BlockType);
        mpData = (BlockType*)realloc(mpData, size);

        // Copying other elements value to new allocated memory
        // using variable's copy method.
        CopyElements(rOther);
      }

      return *this;
    }
```

Like the previous container clearing the container consists of manually calling the destructor of each variable and then freeing the memory. Using the appropriate member of the variable class simplifies this procedure as seen before.

7.4 Common Organizations of Data

Different ways of distributing the data are used in finite element programs. Each of them has its advantages and disadvantages and can be useful for some cases while imposing difficulties to other problems. In this section some of the existing data distributions are explained and their properties are emphasized.

7.4.1 Classical Memory Block Approach

An old standard form of keeping data in memory is an indexed block memory container. Old fortran codes usually use this approach due to the restriction of old fortran compilers. Also some new codes are still using it because of its great performance.

In this approach each category of data is stored in a block of memory. The ordering of data in this block of memory depends on the algorithm uses the data. Some algorithms take one `Node` or `Element` and work over their data and then go to another one. In this case data of each `Node` and `Element` must be stored sequentially to minimize the cache miss while operating over a `Node` or an `Element`. Figure 7.61 shows this alignment of data.

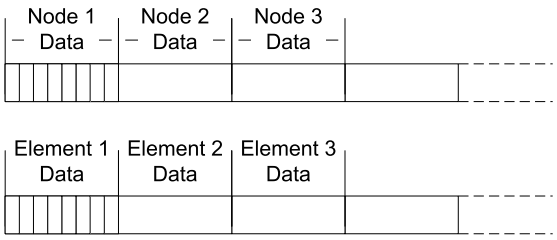


Figure 7.61: Grouping all variables of one `Node` or `Element` in order to reduce the cache miss in nodal and elemental operations.

Some other algorithms take one variable, for example the displacement, and then perform some operations over this variable for all `Node`s or `Element`s. For these algorithms an efficient alignment is to store the values of each variable in different `Node`s or `Element`s sequentially. In this way the cache missing is minimal and the vectorization of the process for parallel computing can be done more effectively. Figure 7.62 shows this alignment of data.

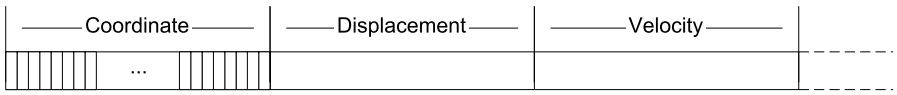


Figure 7.62: Storing values of each variable in different `Node`s or `Element`s sequentially to optimize the data structure for algorithms working with one variable over the domain.

An extension of this alignment is to group the variable components for algorithms working with components separately as can be seen in figure 7.63.

This structure is simple to implement but requires the programmer to know the exact number of variables per node or gauss points and also the size of the buffer which can be difficult to determine

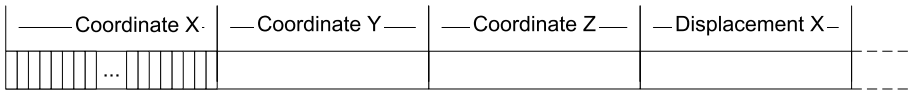


Figure 7.63: Separating the components of each variable for algorithms working with each component separately.

in some cases. By having all these information a block of memory for each group of entities can be allocated and all the data can be stored easily. Also a matrix indexing system can be established to help accessing a certain variable of an entity like a `Node` or an `Element`.

Advantages

- Very easy to program and to use. Normally indexing is used to access inside the data (via a global pointer) and there is no extra pointer to allocate and deallocate so it is easy to create and use and also to clear.
- Very fast and efficient in using system cache. Having all data together makes it easy to keep the system cache full in runtime. This advantage is more apparent when we have a loop over a data for all entities.
- Operation over these data structures can be easily parallelized for shared memory platforms.

Disadvantages

- The structure is rigid for adding new variables. Also having different variables in different `Node`s or `Element`s makes a big unnecessary overhead. For example when we attempt to introduce a new variable to some `Node`s we have to add a new row (or column) to this block. This means that we have to allocate memory equally for this variable in all `Node`s, wether they have that variable or not. This overhead in some codes is not big but for some other codes can be very important.
- This container is homogeneous and hence is not suitable for cases when we need a container to store different data types in it.
- Adding or removing `Node`s or `Element`s causes big reallocation in this data structure, this makes it less interesting for problems where number of `Node`s and `Element`s is changing continuously.

7.4.2 Variable Base Data Structure

This is an step forward from the previous data structure. In this approach the data related to each nodal or elemental variable are stored in a separate array. For example the structure has four arrays to store nodal coordinates, displacements, velocities and accelerations. So in this approach a variable can be added, allocated or removed from memory anytime its needed.

In this approach any algorithm operating over some entities not only has to take an array of entities as its arguments but also needs different arrays of data which are required for its operation. This makes the input and output of method more readable in the code but puts more restriction for some generic methods. However one can create a table of all variables and their names and pass them as an additional argument to guarantee the extendibility of the methods.

Many fortran codes as well as several C and C++ codes use this format for storing their data.

Advantages

This container has a many advantages which makes it popular in finite elements codes.

- Good performance in adding new variables or removing existing ones. In this approach creating new variables or removing some existing ones makes no changes in the global structure and will not affect other parts of the data structure.
- Very fast and efficient in using system cache for algorithms which are oriented to work with variables over the domain.
- Capable of adding new types. This structure can be used to store different types of data without any problem. Also adding new types of variables makes no difficulties because each variable's data are stored in a different array which can have any type of data.
- Easy to program and to use. Creating this data structure is relatively easy with less requirements than for the memory block approach. It is also easy to use as accessing is only via one indexing without any redirection or searching.
- Like the previous approach, the operation over these data structures can be easily parallelized for shared memory platforms.

disadvantages

- Fair performance in removing some entities' data from the container. For problems involving `Node` inserting and removing or `Element` inserting and removing this type of container can introduce a large overhead. To avoid this problem, one can assign a removed flag for entities and update the data structure once, however this method also has its own complexities.
- Not very efficient for using in algorithms working entity by entity. In this structure different data related to one entity can be stored in very far places from each other in memory. This results in cache misses which reduces the performance of the algorithm.

7.4.3 Entity Base Data Structure

In an abstract point of view we can assume this container as the transpose structure of variable base container. In this structure we put all the data related to one entity in the same group. For example all data related to a certain `Node` are stored together. But unlike the memory block there is no guarantee that the blocks of nodal data are sequentially stored in memory.

A rigid but very fast implementation is to make each variable of an entity a member of it. In this way the locality of data is guaranteed, which increases the performance of the application. The problem is the rigidness of this implementation. Each new variable must be added to the entity in programming time. For example `Node` has to have all variables of different problems that the application can solve. For this reason in multi-disciplinary codes this approach cannot be used.Figure 7.64 shows this structure.

Another approach is to allocate a block of memory for the data related to each entity and giving to the entity a reference to its data. A practical way is to define a generic container as the member of each entity where it can store its data. This implementation is more flexible but less efficient because separating the block of data from the entity produces a jump in memory for accessing it and produces cache missing. Figure 7.65 shows this implementation.

Using this structure, algorithms operating over entities just has to take the array of the entities. Because each entity knows how to access its data so algorithm can access its necessary data from

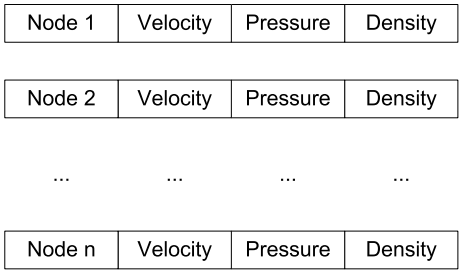


Figure 7.64: In an entity base data structure all data related to one entity can be stored with entity as its members.

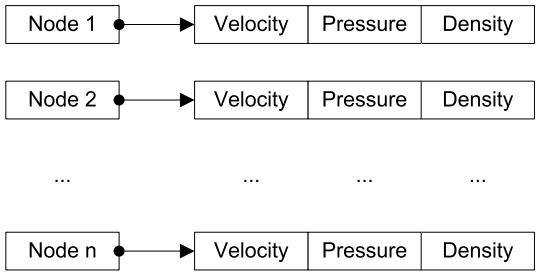


Figure 7.65: Each entity has a reference to its container or block of data.

the entity and there is no need to give its input and output as additional arguments. This reduces the number of arguments while maintaining the extendibility of the code.

Advantages

- Good performance in adding an entity or removing an existing one. The data related to each entity are grouped together separately to other data, so any new group of data related to a new entity can be added without affecting other parts of the data structure. Also removing data related to one entity can be done independently without any problem.

- Separating data in this way makes parallelization of the application over distributed memory architectures easier. In this way there indexing is not necessary to find the data of an entity. This independency is handy for dividing data over machines.

- Good use of cache memory when used in algorithms which operate on entities. The algorithm can access several data of one entity to do its operation and then goes to the next entity. This structure has a good performance because keeps all the data in one block of memory and significantly reduces the cache misses.

Disadvantages

- Fair performance in adding new variables or removing existing ones. In this approach introducing new variables or removing some existing ones requires an entity by entity resizing of data, which makes it less efficient than the previous approach.
- Less efficient in using system cache for algorithms which are oriented to work with variables over the domain. Making a loop over a variable holds in entities produces jumps in memory when going from one entity to another. This make it less efficient in using the cache memory and therefore reduces the performance of the algorithm.

7.5 Organization of Data

In a finite element program several categories of data has to be stored. Nodal data, elemental data with their time histories and process data are examples of these categories. Also in a multi-disciplinary application, `Node`s and `Element`s can be stored in different categories representing domains or other model complexities. In this section the global distribution of data in Kratos will be discussed.

7.5.1 Global Organization of the Data Structure

In previous section some common ways to organize data in finite element application were described and their advantages and disadvantages were discussed. It was seen that both variable base and entity base structure offer good features but for two very different type of algorithms. The first structure is optimized for domain based algorithms and also when some variable has to be added or removed from data structure. While the second structure is better for entity based algorithms and is more flexible for adding or removing entities.

Kratos is designed to support an elemental-based formulation for multi-disciplinary finite element applications and also started with mesh adaptivity as one of its goals. So the entity based data structure becomes the best choice. First because elemental algorithms are usually entity based and can be optimized better using this type of structure. The second reason is the good performance and flexibility this structure offers, in order to add or remove `Node`s and `Element`s. Beside this entity base structure Kratos also offers different levels of containers to organize and group geometrical and analysis data. These containers are helpful in grouping all the data necessary to solve some problems and for simplifying the task of applying a proper algorithm to each part of the model in multi-disciplinary applications.

Nodal, elemental and conditional data containers are the basic units of this entity base structure. In Kratos each `Node` and `Element` has its own data. In this manner an `Element` can access easily the nodal information just by having a reference to its Node and without any complications. `Properties` also is a block of this structure as a shared data between `Element`s or `Condition`s. Figure 7.66 shows these basic units and their relations to different entities.

Separate containers for `Node`s, `Properties`, `Element`s and `Condition`s are the first level of containers defined in Kratos. These containers are just for grouping one type of entity without any additional data associated to them. These containers can be used not only to work over a group or entities but also to modify their data while each entity has access to its own data. These containers are useful when we want to select a set of entities and process them. For example giving a set of `Node`s to nodal data initialization procedure, sending a set of `Element`s to assembling functions, or getting a set of `Condition`s from a contact procedure. Figure 7.67 shows these containers and their accessible data.

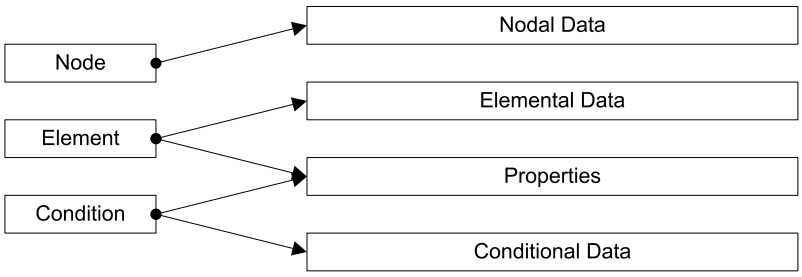


Figure 7.66: Nodal, elemental and conditional data containers with properties are the basic units of Kratos data structure.

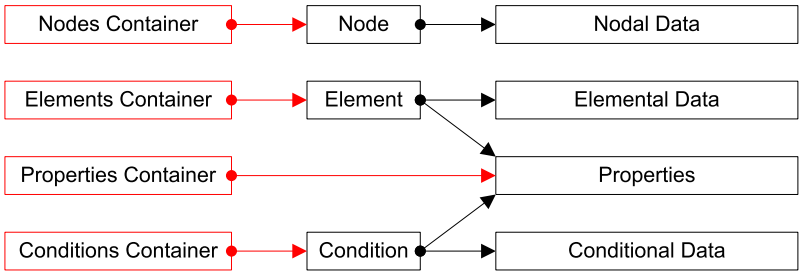


Figure 7.67: Separate containers for `Node`s, `Properties`, `Element`s and `Condition`s can be used to group each type of entities and then process themselves or their accessible data.

`Mesh` is the second level of abstraction in the data structure which hold `Node`s, `Element`s and `Condition`s and their `Properties`. In other word, `Mesh` is a complete pack of all type of entities without any additional data associated with them. So a set of `Element`s and `Condition`s with their `Node`s and `Properties` can be grouped together as a `Mesh` and send to procedures like mesh refinement, material optimization, mesh movement or any other procedure which works on entities without needing additional data for their processes. Figure 7.68 shows `Mesh` with its components.

The next container is `ModelPart` which is a complete set of all entities and all categories of data in the data structure. It holds `Mesh` with some additional data referred as `ProcessInfo`. Any global parameter related to this part of the model or data related to processes like time step, iteration number, current time, etc. can be stored in `ProcessInfo`. `ModelPart` also manages the variables to be hold in `Node`s, `Element`s and `Condition`s. For example all the `Node`s belonging to one `ModelPart` sharing the nodal variables list hold by it. From another point of view `ModelPart` is the nearest container to the domain concept in the multi-disciplinary finite element method. Figure 7.69 shows the `ModelPart` with its components.

In the first implementation, `ModelPart` was able to keep the history of data and also the `Mesh` if it is changing. But in practice this capability became the bottleneck of Kratos performance and was also considered to be unnecessary for our problems. So this feature was removed from `ModelPart`. However each `ModelPart` still can hold more than one `Mesh` which comes from the first implementation and can be used for representing the partitions in parallel computation.

Finally `Model` is a group of `ModelPart`'s and represents the finite element model to be analyzed.

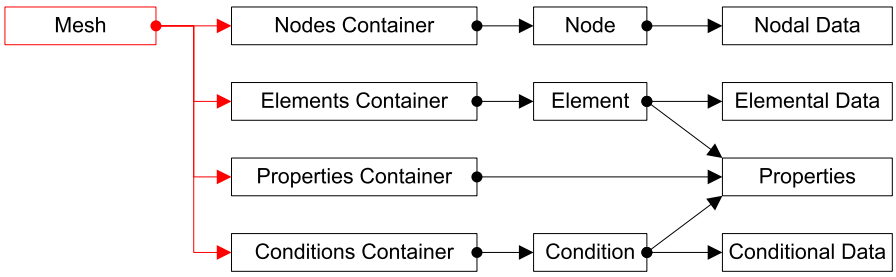


Figure 7.68: `Mesh` is a complete pack of all types of entities without any additional data associated with them.

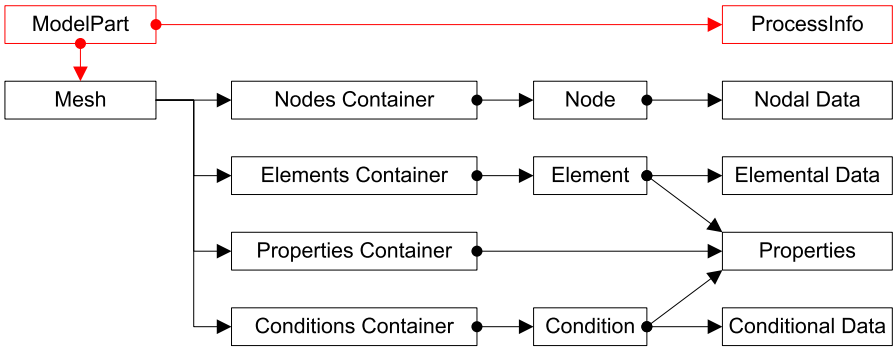


Figure 7.69: `ModelPart`holds `Mesh` with some additional data referred as `ProcessInfo`

It can be useful for some procedures that requires the whole data structure like saving and loading procedures. As processes in Kratos use `ModelPart` as their work domain, this container is not implemented yet but it is necessary to complete the data structure of Kratos.

Spatial containers are separated so can be used just when they are needed. This strategy also allows Kratos to use external libraries implementing general spatial containers like Approximate Nearest Neighbor (ANN) library [74].

7.5.2 Nodal Data

The first implementation of Kratos had a buffer of data value containers to hold all the nodal variables. This nodal container was very flexible but with considerable memory overhead for nonhistorical variables. For example for saving two time steps in history (a buffer with size 3) there were two redundant copies of all nonhistorical variables in memory as shown in figure 7.70. It also had fair access performance due to the searching process of the container. All these made us to redesign the way data are stored in `Nodes`.

The new structure is divided into two different containers: nodal data and solution step nodal data. Figure 7.71 shows this nodal data structure. A data value container is used for the nodal data (no historical data) and a variables list container is used for the solution step nodal data (historical data). In this way the memory overhead is eliminated because no redundant copy is

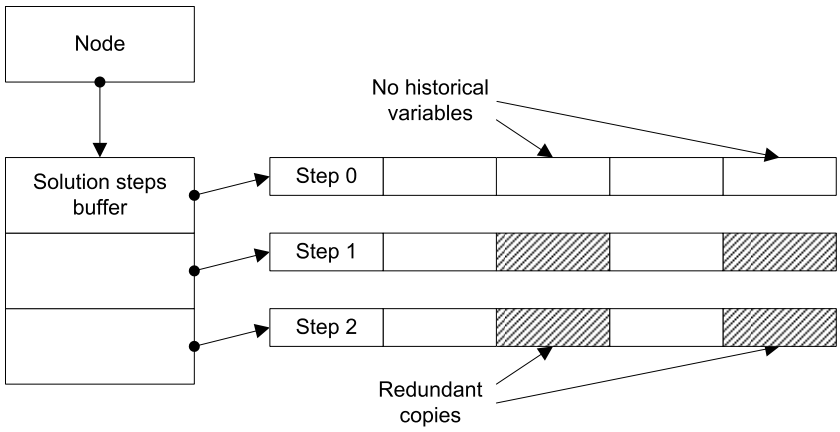


Figure 7.70: Using buffer for all variables results memory overhead due to redundant copies of no historical variables.

produced. Also accessing to historical variables is much faster than before due to the indirection process of accessing in the variables list container instead of the searching process in the data value container. This structure offers good performance and also is memory efficient but is slightly less robust and somehow less flexible to use.

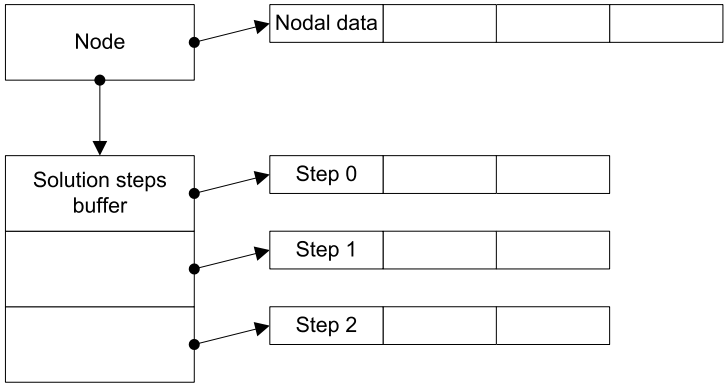


Figure 7.71: The first improvement is dividing nodal data structure into two different containers, nodal data (no historical data) and solution step nodal data (historical data).

Using the variables list container for historical variables, requires the user to define its historical variables in order to construct the nodal data container. For example a fluid application must define velocity and pressure as its historical variables at program startup. The rest of variables can be added any time during the program execution as a no historical variable, but not as a historical one.

The first implementation of this structure was done by creating a buffer of the variables list container to reduce the implementation task and test it in a real problem. After obtaining successful results it was the time to optimize it more. The buffer of variables list containers produces several

jumps in memory which reduces its cache efficiency. Also some mesh generators like TetGen [93, 7] require an array of nodal data as the argument to make the interpolation. For these reason in the current structure the buffer is moved inside the solution steps container. In this way the cache misses is reduced and the data array can be given to other application without any conversion. Figure 7.72 shows this structure.

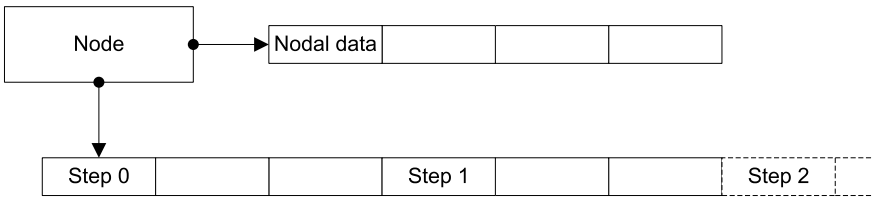


Figure 7.72: The current structure allocates all buffer data in a block of memory to reduce the cache misses produced by memory jumps and also to provide a compatible data with other libraries.

In the first structure all the data were stored in the same container. So there was one place to store them and also one place to recover them. Dividing the nodal data in two categories also changes the access interface to data. Now the user has to know were to put each variable, and more important, where to retrieve them afterward. A sophisticate interface is needed in order to provide a clear and complete control over these two categories of data. In general, three type of accessing methods are necessary:

- Methods for accessing only historical data. These methods guarantee to give the value of the variable if and only if it is defined as a historical variable and produce error if it is not defined. These methods are very fast because they do not need to search in the data value container and also give error for logical errors in the code.
- Methods for accessing only no historical data. Another set of methods are implemented to give access only to no historical data. Due to the flexibility of the data value container any variable can be added at any time as a nodal variable using these methods.
- Hybrid accessing also can be done using another set of methods. These methods try to find the variable in the solution steps container and if it does not exist they provide access to the nodal data container. These methods are helpful for accessing to some input variables that may come from input files as nodal data, or variables which are calculated in another domain and stored as a solution step variable. For example temperature for structural problem can be a parameter coming from the input data or calculated by the thermal elements and stored in `Node`s. This method guarantees the access to the proper temperature stored at each `Node`.

The following methods are implemented to provide above access ways to nodal data:

`GetSolutionStepValue` Takes a variable or a variable's component and the solution step index and returns its value in solution step data if exists, otherwise sends an error. Solution step index starts from 0 for the current step and then increases for past steps. For example 1 is for the previous step, 2 is for one step before the previous one, etc. An overloaded version accept only the variable or a variable's component and returns its current value. In all overloaded versions accessing to a variable which is not declared in the solution steps variables list produces an exception.

`GetValue` There are different overloaded versions of this method. One with the requested variable as its argument, gives an access to nodal data without looking to the solution steps data. Giving a solution steps index as an additional argument makes it look into nodal data and if it does not exist takes it from the solution steps container. It will find any existing value in the nodal data structure for given variable but searching in data makes it slow in accessing to the solution steps data.

7.5.3 Elemental Data

Another basic unit of Kratos data structure is the elemental data. Elemental data is divided into three different categories:

properties All parameters that can be shared between `Element`s. Usually material parameters are common for a set of `Element`s, so this category of data is referred as properties. But in general it can be any common parameter for a group of `Element`s. Sharing these data as properties reduces the memory used by the application and also helps updating them if necessary.

data All variables related to an `Element` and without history keeping. Analysis parameters and some inputs are elemental but there is no need to keep their history. These variables can be added any time during the analysis.

historical data All data stored with historical information which may be needed to be retrieved. Historical data in integration points are fall in this category. These data must be stored with a specific size buffer.

As mentioned above `Properties` are shared between `Element`s. For this reason the `Element` keeps a pointer to its `Properties`. This connection lets several `Element`s to use the same `Properties`.

A `DataValueContainer` holds no historical data in the `Element`. Using a `DataValueContainer` provides flexibility and robustness which is useful in transferring elemental data from one domain to another. It is important that these no historical data are not the most used during the analysis and flexibility here is more critical than performance. On the contrary, historical data are more used during the analysis and efficiency in accessing to them is more critical than their flexibility. These data are specified by formulation and other processes do not change them. For this reasons the `Element` do not provide any container for them and these containers can be implemented by element developers. In this way, the customized container will be more efficient and the overhead of any generic container will be eliminated.

7.5.4 Conditional Data

Conditional data is very similar to elemental data and is also divided into three different categories:

properties As for `Element`s, all parameters that can be shared between `Condition`s is referred as properties. Again sharing these data as properties reduces the memory used by the application and also helps updating them if it is necessary.

data All variables related to the `Condition` and without history keeping. Analysis parameters and some inputs are different for each `Condition` but there is no need to keep their history. This variables can be added any time during the analysis.

historical data All data stored with set of its history to be retrieved. Historical data in integration points are in this category. These data must be stored with a specific size buffer of their previous values to be used later.

`Condition` like `Element` keeps a pointer to its `Properties` which is shared by other `Conditions` or `Elements`.

The `Condition` has a `DataValueContainer` as its member to hold all data related to `Condition` without keeping its history. Any `Condition` derived from this class can use this container to hold its data without any additional implementation. This base class also provides an standard interface to these data which make it helpful for transferring some data from one `Condition` to another, for example in the interaction between two domains.

For `Conditions`, historical data is considered to be an internal data because is very related to its formulation and usually is used only by formulation inside and not from outside. So to minimize unnecessary overhead and also to increase the performance, no general container is provided for historical data and each `Condition` has to implement one for itself if necessary.

7.5.5 Properties

As mentioned before `Properties` is a shared data container between `Elements` or `Conditions`. In finite element problems there are several parameters which are the same for a set of `Elements` and `Conditions`. Thermal conductivity, elasticity of the material and viscosity of the fluid are examples of these parameters. `Properties` holds these data and is shared by `Elements` or `Conditions`. This eliminates memory overhead due to redundant copies of these data for each `Element` and `Condition` as can be seen in figure 7.73.

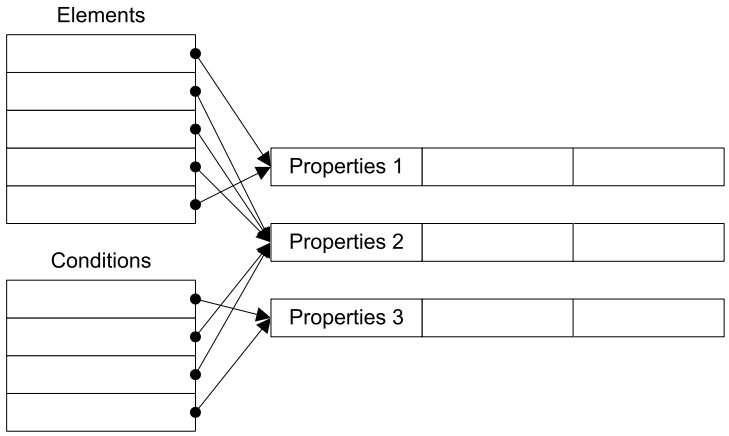


Figure 7.73: Different `Elements` or `Conditions` use `Properties` as their share data container. This avoids redundant copies of data in memory.

It can be seen that changing any data in `Properties` will affect all `Elements` or `Conditions` sharing it. This feature can also be useful in situations when a common parameter is changing during the analysis. The parameter can be changed only in `Properties` and each `Element` or `Condition` will get the new value by accessing to `Properties`. In this way there is no need to update all elemental and conditional value for this data each time its changing.

Properties also can be used to access nodal data if it is necessary. It is important to mention that accessing the nodal data via Properties is not the same as accessing it via Node. When user asks Properties for a variable data in a Node, the process starts with finding the variable in the Properties data container and if it does not exist then get it from Node. This means that the priority of data is with the one stored in Properties and then in Node. For example considering that TEMPERATURE is stored in Properties as a material temperature with value 24.4 and also there is a TEMPERATURE nodal data stored in Node with value 79.3. Now getting the value of TEMPERATURE variable from Properties gives 24.4 while getting it from Node gives 79.3

7.5.6 Entities Containers

Let us go one level higher in the Kratos data structure. The next level consists of four entities containers:

- Nodes Container
- Properties Container
- Elements Container
- Conditions Container

In a finite element program, there are several procedures which take a set of entities and operate over them or their data. These containers are created to help users in grouping a set of entities and work with them. For example to put all Nodes in the boundary in a Nodes container and change some of their data in each step. As mentioned before, each entity has access to its data, so having a set of entities in a container also gives access to their data which make these containers more useful in practice.

Another use of these containers is finding an entity by its index. Indexing is an standard way of identifying entities in finite element programs. For example Nodes have their indices to be identified by them. The index of each Node is given by the user as input and can be consecutive or not. A user will use these indices later to define the elemental connectivity. In time of creating Element, it is necessary to find the Node with a given index and give its pointer to the Element. Supporting the indexing system and providing a searching mechanism is very useful for implementing such a processes in a simple and efficient manner.

According to all uses mentioned before, a suitable container for holding entities must provide the following features:

Sharing Entities There are some situations when an entity may belong to more than one set of entities. For example a boundary Node belongs to the list of all Nodes and also to the list of boundary Nodes. So the Nodes container has to share some its data with other Nodes containers. In general, sharing entities is an important feature of these containers.

Fast Iterating As mentioned before, one important use of these containers is to collect some entities and pass them to some procedures. Usually these procedures have to make a loop over all the elements of a given container and use each element or its data in some algorithms. So these containers must provide a fast iterating mechanism in order to reduce the time of element by element iteration.

Search by Index Finding an entity by an index is a usual task in finite element programs. So entities containers must provide an efficient searching mechanism to reduce the time of these tasks.

Method Name	Operation
NumberOfNodes	Returns the number of Nodes in the Mesh.
AddNode	Add given Node to its Nodes container.
pGetNode	Returns a pointer to the Node with the given identifier.
GetNode	Returns a reference to the Node with the given identifier.
RemoveNode	Removes the Node with given Id from the Mesh.
RemoveNode	Removes the given Node from the Mesh.
NodesBegin	Returns a Node iterator pointing to the beginning of the Nodes.
NodesEnd	Returns a Node iterator pointing to the end of the Nodes container.
Nodes	Returns the Nodes container.
pNodes	Returns a pointer to the Nodes container.
SetNodes	Sets the given container as is Nodes container.
NodesArray	Returns the internal array of Nodes.

Table 7.1: Interface of Mesh for accessing Nodes

Sharing entities is the first feature to be provided by containers. Holding pointers to entities and not entities themselves can solve this problem. Different lists can point to the same entity without problem. Using an *smart pointer* [37] instead of normal pointer increases the robustness of the code. In this way entities which are not belong to any list anymore will be deallocated from memory automatically. So a container of smart pointers to entities is the best choice for this purpose.

As mentioned above arrays are very efficient in time of iterating. So using an array to hold pointers to entities can increase the iteration speed. In contrary, trees are very slow in time of iterating but efficient for searching by index hence using them can increase the searching performance of the code. A good solution to this conflict can be an ordered array. It is fast in iteration like an array and also fast in searching like a tree. Its only draw back is that its less robust and constructing it can take considerable time depending on the data. For example constructing an array of Nodes with Nodes given by inverse order can take a very long time. Fortunately most of the time the entities are given in the correct order and this eliminates the constructing time overhead for these containers. Also, for the worst cases a buffer of unordered data can significantly reduce the construction overhead. So an ordered array can fit properly into our problem.

PointerVectorSet is a template implementation of an ordered array of pointers to entities. This template is used to create different containers to hold different type of entities.

7.5.7 Mesh

The next level in Kratos' data structure is Mesh. It contains all entities containers mentioned before. This structure makes it a good argument for procedures that work with different entities and their data. For example an optimizer procedure can take a Mesh as its argument and change geometries, nodal data or properties. Mesh is a container of containers with a large interface that helps users to access each container separately.

First of all Mesh provides a separate interface for each type of entity it stores. Tables 7.1, 7.2, 7.3 and 7.4 show its interfaces for handling different components.

Mesh holds a pointer to its container. In this way several Meshes can share for example a Nodes or an Elements container. This helps in updating Meshes of different fields in multidisciplinary applications but over the same domain. Figure 7.74 shows this ability of sharing components between Meshes.

Method Name	Operation
`NumberOfProperties`	Returns the number of properties in the `Mesh`.
`AddProperties`	Add given properties to its properties container.
`pGetProperties`	Returns a pointer to the properties with the given identifier.
`GetProperties`	Returns a reference to the properties with the given identifier.
`RemoveProperties`	Removes the properties with given Id from the `Mesh`.
`RemoveProperties`	Removes the given Properties from the `Mesh`.
`PropertiesBegin`	Returns the begin iterator of the properties container.
`PropertiesEnd`	Returns the end iterator of the properties container.
`Properties`	Returns the properties container.
`pProperties`	Returns a pointer to the properties container.
`SetProperties`	Sets the given container as is properties container.
`PropertiesArray`	Returns the internal array of properties.

Table 7.2: Interface of `Mesh` for accessing properties

Method Name	Operation
`NumberOfElements`	Returns the number of `Elements` in the `Mesh`.
`AddElement`	Add given `Element` to its `Elements` container.
`pGetElement`	Returns a pointer to the `Element` with the given identifier.
`GetElement`	Returns a reference to the `Element` with the given identifier.
`RemoveElement`	Removes the `Element` with given Id from the `Mesh`.
`RemoveElement`	Removes the given `Element` from the `Mesh`.
`ElementsBegin`	Returns the begin iterator of the `Elements` container.
`ElementsEnd`	Returns the end iterator of the `Elements` container.
`Elements`	Returns the `Elements` container.
`pElements`	Returns a pointer to the `Elements` container.
`SetElements`	Sets the given container as is `Elements` container.
`ElementsArray`	Returns the internal array of `Elements`.

Table 7.3: Interface of `Mesh` for accessing `Elements`

Method Name	Operation
`NumberOfConditions`	Returns the number of `Conditions` in the `Mesh`.
`AddCondition`	Add given `Condition` to its `Conditions` container.
`pGetCondition`	Returns a pointer to the `Condition` with the given identifier.
`GetCondition`	Returns a reference to the `Condition` with the given identifier.
`RemoveCondition`	Removes the `Condition` with given Id from the `Mesh`.
`RemoveCondition`	Removes the given `Condition` from the `Mesh`.
`ConditionsBegin`	Returns the begin iterator of the `Conditions` container.
`ConditionsEnd`	Returns the end iterator of the `Conditions` container.
`Conditions`	Returns the `Conditions` container.
`pConditions`	Returns a pointer to the `Conditions` container.
`SetConditions`	Sets the given container as is `Conditions` container.
`ConditionsArray`	Returns the internal array of `Conditions`.

Table 7.4: Interface of `Mesh` for accessing `Conditions`

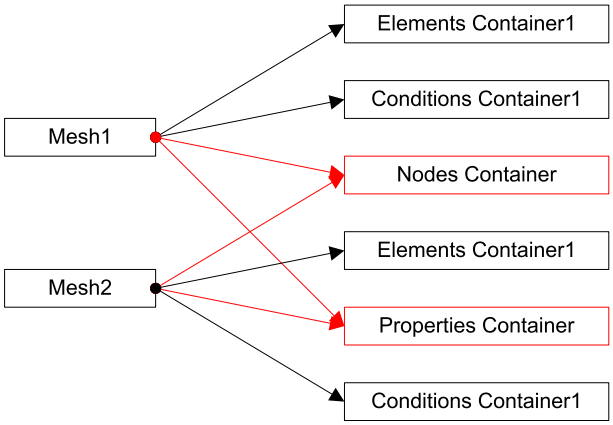


Figure 7.74: Different `Meshes` can share their entities' containers.

7.5.8 Model Part

`ModelPart` is created with two different tasks in mind. The first task is encapsulating all entities and data categories of Kratos which makes it useful as an argument of global procedures in Kratos. The second task is managing the variables lists of its components.

`ModelPart` can hold any category of data and all type of entities in Kratos. It can hold several `Meshes`. Usually just one `Mesh` is assigned to it and used in the computations, however this ability can effectively used for partitioning the model part and send it for example to the parallel processes. Beside holding different `Meshes`, it also stores the solution information encapsulated in the `ProcessInfo` object. Figure 7.75 shows the structure of `ModelPart`.

`ProcessInfo` holds not only the current value of different solution parameters but also stores their history. It can be used to keep variables like time, solution step, non linear step, or any other variable defined in Kratos. Its variable base interface provides a clear and flexible access to these data. `ProcessInfo` uses a linked list mechanism to hold its history as shown in figure 7.76.

`ModelPart` uses pointers to its `Meshes`. In this way it can share them with any other model parts if necessary. A typical use of this feature is defining two different domains over the same

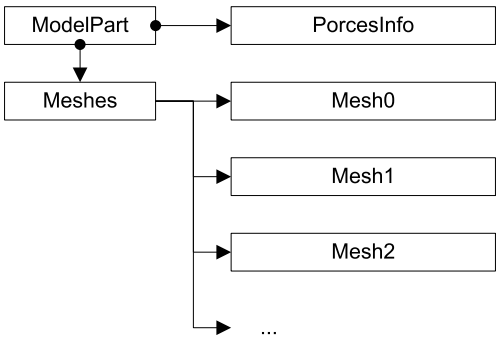


Figure 7.75: `ModelPart`'s structure.

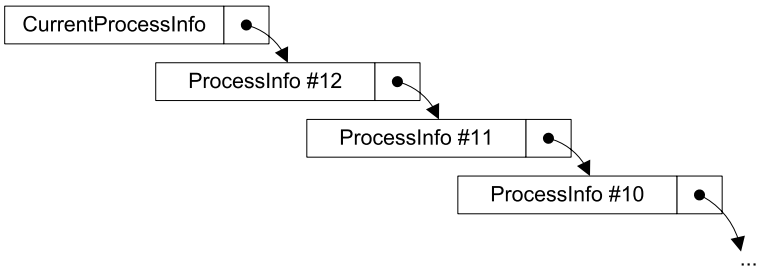


Figure 7.76: `ProcessInfo`'s linked list mechanism for holding history of solution.

`Meshes`. Figure 7.77 shows this sharing mechanism.

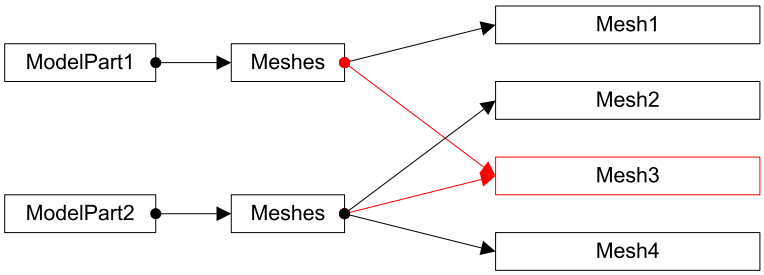


Figure 7.77: `ModelPart` can share its `Meshes` with other model parts.

`ModelPart` manages the variables lists of its components. In section 7.3.3 the mechanism of the variables list container has been described, where we also mentioned that a shared variables list specifies the data which can be stored in them. `ModelPart` holds this variables list for all its entities. In other words, all entities belonging to a model part sharing the same list of variables. For example all `Nodes` in `ModelPart` can store the same set of variables in their solution steps container. It is important to mention that this variables list is assigned to the entities which belong to the model part and is not changed when that model part share them with other model parts. Figure 7.78 shows this scheme.

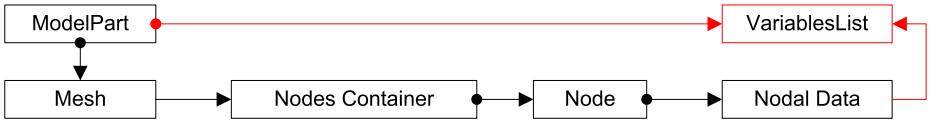


Figure 7.78: `ModelPart` manages the variables list for its entities.

7.5.9 Model

`Model` is the representation of whole physical model to be analyzed via the FEM. The main purpose of defining `Model` is to complete the levels of abstraction in the data structure and a place to gather

all data and also hold global information. This definition makes it useful for performing global operations like save and loading. It holds references to model parts and provides some global information like the total number of entities and so on. It can be seen that a `ModelPart` created over given model can do most of these operations by itself. This was the reason that `Model` itself has not been used yet in Kratos. Its practical use would be at the time of implementing the serialization process or other global operations.