Accelerated Square Array

Attila, Márton

Disclaimer

This is the translation of the thesis for the Accelerated Square Array, from hungarian. It only contains the main scientific part.

Abstract

The storage of data is an essential part of modern computational systems. The storage of data usually happens in data structures, the speed of which, fundamentally effect the speed of a software or system of softwares. Modern computer science offers several tools, complex data structures, among which one can find a fitting solution for almost any problem. On top of direct usage, data structures may be used for implementing more complex data structures.

In this work, I would like to show a new alternative, in a subgroup of data structures that is an essential part of almost all complex program and data structure, yet only a small fraction of it's members is used. The arrays, or more generally, data structures with constant time index based access is present in almost all application despite the speed of insertion and deletion in these being proportional to the total size.

In this work, I would like to introduce the Accelerated Array, which similarly to classical Arrays, is capable of index based access of elements in constant time, but also implements, insertion and deletion at a guaranteed speed of the square root of the total size.

In this work, I would like to show the implementation, and the proof of the aforementioned capabilities, and compare it to the currently existing alternatives.

Keywords: data structure, array, data storage, database, data change

Table of contents

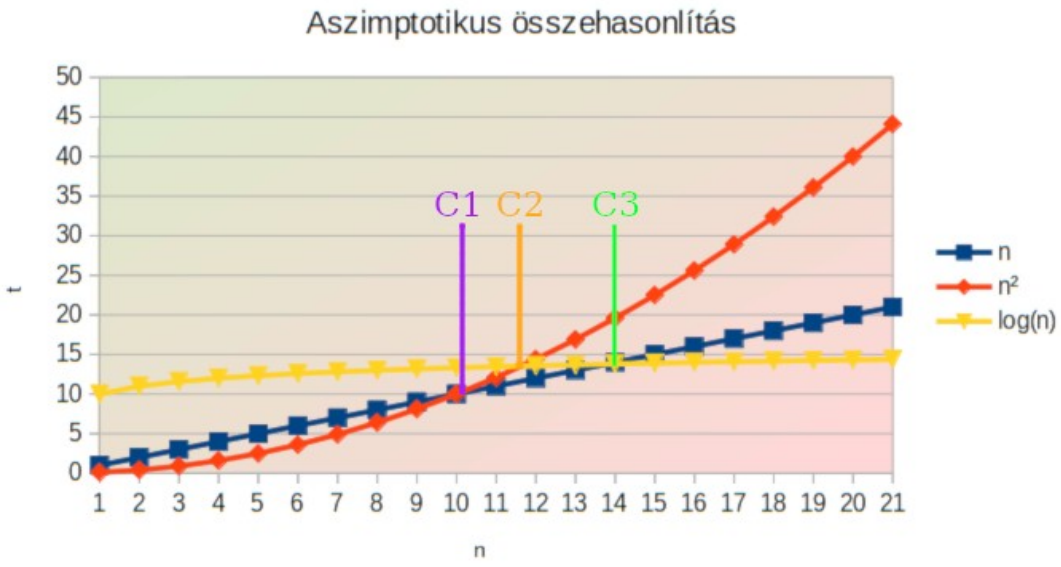
Phrases

Access :	Read and write
Mutation :	Insertion and deletion

1 State of affairs

Currently there are many data structures, with different advantages and disadvantages. For the sake of comparing and using them, knowledge of them is necessary. To discuss different advantages, and measurements, I first msut introduce fundamental terms.

Since speeds are strongly influenced by implementation and physical architecture, measuring naive implementations alone would not give a complete picture. Furthermore, the theoretical limitations of an algorithm or data structure are also an important factor. In order to discuss these in a mathematically rigorous way, it is worth considering how a data structure or algorithm behaves when the number of input or stored elements exceeds all limits, i.e. when it approaches infinity. This in itself would carry only two possibilities, either it goes to infinity or to some constant. In order to compare the rates to infinity, we need to introduce a mathematical notation method that can establish some kind of relation, a hierarchy, between the functions to infinity.



1. Figure 1: Comparison of functions

A function f can be said to be larger than another function *g* at infinity if, even for an arbitrarily large positive constant coefficient *c*, there exists a threshold index *n0* over *n* variables at infinity such that for $n > n0$: $c * f(n) > g(n)$. We denote this by $g = O(f)$. This is the notation for the large O. It may also be the case that, when increasing *n* to infinity, there

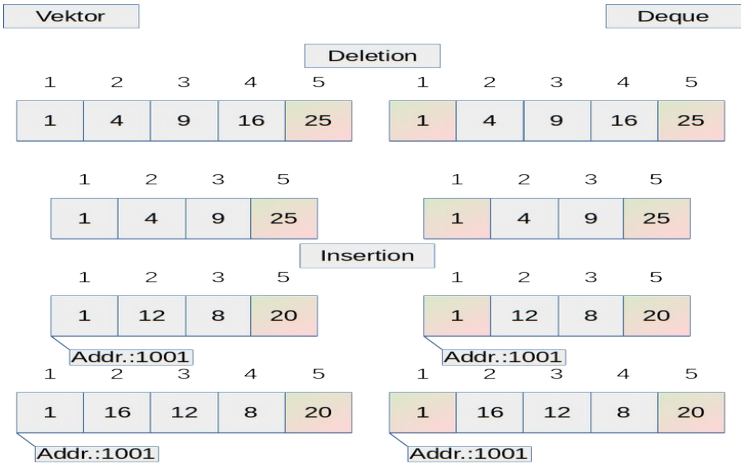
exists a threshold index *n0* from which, for *n > n0: c * f*(*n*) < g(*n*). Then *f* estimates *g from* below. We denote this by *g* = Ω(*f*). If both hold, *g* = Θ(*f*).

Accordingly, the quadratic function in Figure 1 gives an upper estimate of the other two, the linear function *from C1* and the logarithmic function *from C2*. Accordingly, they underestimate the quadratic function from the same thresholds. The linear function *from C3* estimates the logarithmic *from* above. Accordingly, the logarithmic function *from C3* estimates the linear from below. In addition, each function estimates itself from below and from above, since with a constant multiplier greater than one, all elements are greater than the original value, and with a constant multiplier less than one, all elements are less than the original value. The foundation for the next chapter, which summarizes existing technologies, is laid in the book New Algorithms by Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein.

1.1 Static and Dynamic Arrays

Currently, among sequential Data Structures, the Static Array and its dynamic variants (Vector and Deque) are almost the only ones. A Static Array is an extremely simple data structure that stores its elements in a sequential order, which must be of the same size. The memory address of an element with a given index can be obtained by adding the size of an element to the start of the Static Array (start of the first element), multiplied by the index sought. For this, we need a 0-based addressing, which means that the index of the first element is 0.

Dynamic Arrays extend this by allowing insert and delete operations. Dynamic Arrays simply store data in consecutive locations, taking up more space than necessary, and move data elements in case of mutation. In the case of a Vector, the logical end, in the case of a Deque, the nearer end of the Data Structure. Constant-time index-based access is a very big advantage. It is often worth it even though the mutation is very slow.



2. Figure 2: Vector and Deque

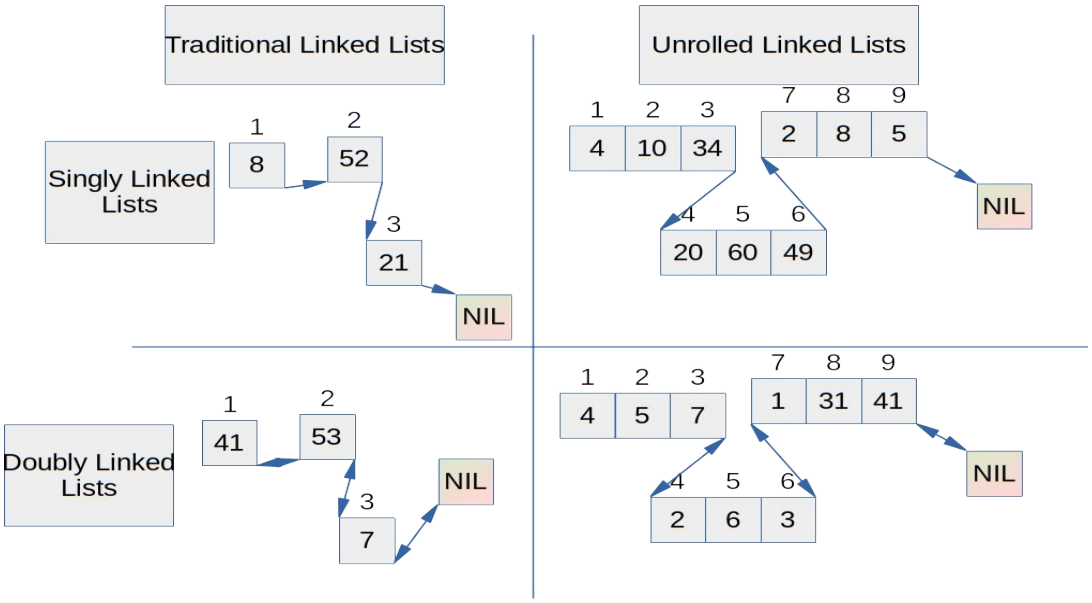
Figure 2 shows the physical location of growth and shrink for Vectors and Deques. The worst case of index-based access is also O(1), while the worst and average case of mutation is Θ(*n)*. The worst case of Deque is to move only half of the elements, while the worst case of a vector (insertion in front of the first element or deletion of the first element) is to move all elements. The main advantage of the Vector is that it can be referred to by a fix address, because of the fixed position of the first element. At the same time, an important factor is that it is possible to insert and delete at constant time at one end of the Vector and at both ends of the Deque, if memory reallocation is not used.

It is reasonable to ignore it, because when reallocation happens, usually twice the current size is requested. Thus as the size of the Deque approaches infinity the chance per element of having to reallocate memory is 1 / *n*,. The work required(copy) is *n*, so the expected work is (1 / *n*) * *n*, which is Θ(1). However, it is also a possibility that in a modern system a nearly full Deque is reallocated by a background thread before the resizing can slow down the program.

Dynamic Arrays are also used to build more complex data structures, in addition to their direct use. Thus, even though modern databases use trees (mostly an improved version of a B Tree, like a B+ Tree), they still depend indirectly on the speed of the sequential Data structures used.

1.2 Traditional and Unrolled Lists

Linked lists are index-based data structures that are built up from nodes. These nodes consist of a data element and at least one pointer to the next element. Nodes may also contain a pointer to a previous element. In this case we are talking about a doubly linked list. If there are only pointers pointing forward, it is a singly linked list. The end of the list is indicated by a null pointer pointing to the invalid address 0. In a linked list, to reach a given index, the list must be traversed to that index. This is an O(*n)* time operation. If the particular element *x* we want to insert after has already been reached as part of another operation, then the insertion is constant time. To do this, we reserve a new node with the data element we want to store, change the new node's pointer to the next element to *x*, and then change *x's* forward pointer to the address of the newly created node. In the case of a double linked list, the backward pointers are also changed according to the same reasoning as before, to reflect the new order. The deletion is also a constant-time operation, ignoring the reach, if the node before the node to be deleted is noted, or if we are working with a double linked list. In this case, the operation only requires that we adjust the pointers to the new state. With linked lists, there is a possiblity of storing more than one element in a node. Then it is an unrolled list. Vector or Deque is usually used for this. In addition, chained lists may have other special solutions, such as guard elements instead of null pointers (NIL) to indicate a unique list end or to indicate a "penultimate" element. In addition, problems that logically require circular data storage may be described by circular linked lists, where the last element is followed by the first. Figure 3 shows the options for Chaining by Direction and Expression, using non-circular examples terminated by a null pointer.



3. Figure 3: Linked List types

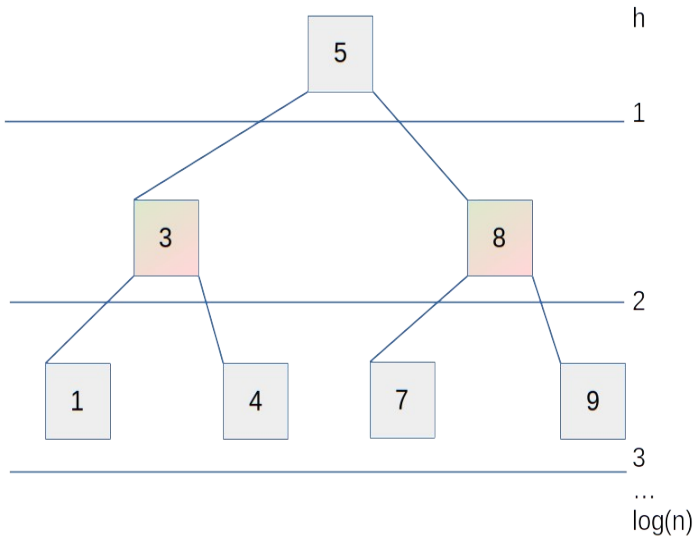
An important use of linked lists is that they can be used to implement trees. This is done by allowing the data member itself to be a linked list. Linked lists are usually intended to be used directly for tasks where sequential or possibly reverse entry is likely, and insertions and deletions are frequent.

1.3 Red-Black-Tree

The Red-Black Tree is a nearly balanced binary search tree. To examine the Red-Black Tree, we first need to understand its meaning. Here I will refer to rooted trees as "trees". Tree data structures are data structures that store their elements in a recursively branched manner. The first element is called the root of the tree and the elements themselves are called nodes. They are usually implemented by linked lists where the data member of the links can itself be a linked list. Accordingly, from a given nodes we can go in at least 2 directions, so after n steps we can reach at least 2^n vertices. This is exponentially better than the *n* linear time of linked lists. The nodes to which we can go from a given node *x are* called the children of node *x*, the number of children is called the degree number of the node. If all nodes have the same number of children, this is the degree of the tree. Trees with the degree of „2” are called binary trees.

The exponentially better speed is only present, if the tree is sufficiently balanced. By the height of a tree, we mean how many steps the furthest node is from the root. A leaf is a node that has no children but stores data. A tree is said to be balanced if all leaves are either the same distance from the root as the furthest leaf, or at most 1 leaf closer. A tree is trivially balanced when it is created empty. There are then several methods for upholding the balance during mutation. The height of an n-element balanced tree is lg(*n*), where the base of the logarithm is the degree of the tree.

Search trees are trees that store some sort of clearly ordered elements. The elements are ordered so that, for 2 degrees, the left child of a node is always smaller, and the right child is always larger or equal. In a balanced search tree, finding a given search value takes O($\log_a$ *(n)*) time, since $\log_a$ *(n)* is the maximum distance from the root. When searching for an item, we start from the root and check whether the node being checked for, is larger than the item currently being searched for. If larger, we move on to the right child, if smaller, we move on to the left, if equal, we have found it. In this way we go through the whole tree. If we arrive at a leaf and it does not match the element we are looking for, the element we are looking for is not in the tree. Higher degree search trees are described in section 1.3, under B-trees. If the tree is not perfectly balanced, but the degree of imbalance is limited, the tree is considered to be nearly balanced.



4. Figure 4: Red Black tree

The Red-Black Tree is a near-balanced binary search tree that achieves balance by using logarithmic time overhead and 1 bit of overhead per node, which indicates whether the vertex is red or black.

The Red Black Tree maintain some intrinsic qualities that help to maintain a near balance. These are:

- All leaves are either red or black
- The roots of the tree and all the leaves are black.
- The children of all red nodes are black
- From the root to each leaf there are the same number of black nodes.

Figure 4 shows a Red Black Tree fulfilling these requirements. These are maintained by recolouring and, if necessary, rotating. The Red Black Tree is able to maintain the property that the node furthest from the root is at most twice as far away as the tip nearest the root.

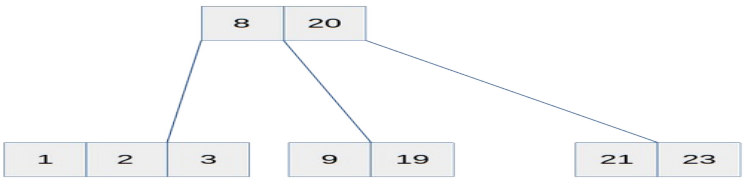
The insertion-deletion and search operations, together with the balancing operations of the Red Black Tree, are also $\Theta(\log_2 n)$ in time.

1.4 B trees

B-trees are the most common complex data structures used to implement data collections where insertions and deletions are frequent. B-trees are balanced search trees that can store more than 2 children per nodes.

The nodes contain the parent's identifier, the children's identifier, and values separating the children, called keys. The children are sorted in order and the key values are placed between them. The key values constrain the values of the child to the left of them from the above and the child to the right of them from below. The bottom level of B trees has leaves that store data (in B+ trees, data is stored only here), and the top level is the root of the tree. Essentially, all operations start from here, as in binary and other search trees, but can be flattened by increasing the number of elements stored in a node, and for operations with time $O(\log(n))$, the logarithm is based on half of the number of elements in a node. B-trees are also faster in practice than binary search trees because they store data close together, so that for a given RAM access, usually 64 bytes in size, the processor will load data that it did not initially request but is likely to need later. This feature becomes even more of an advantage when loading data from an HDD, as much larger memory units are loaded in any case, which the B-tree can take advantage of. The number of degrees at the top of a B tree is not constant, but varies over a range. This range is usually between one and half the maximum number of degrees for a tree, except in the root, where it is allowed to be less than this. The number of

keys in a vertex is one less than the number of children. The internal content of the vertices, both for keys and for children indicators, is provided by Dynamic Arrays.



5. Figure 5: Simple B tree

In the case of B Trees, it solves the balancing problem by keeping the number of degrees of peaks within limits and managing the violation of these limits. If a node remains below the maximum number of degrees after the desired insertion, the insertion of the corresponding element can be performed, in the corresponding leaf, and no further action is required. If a node would contain more elements than allowed during insertion, it is cut in half and the middle element is inserted into the parent. It will then be inserted in the correct place, among the keys, and will become the separator key between the 2 new child nodes. If this insertion into the parent brings the parent above the allowed number of degrees, a split is also required here, with a similar separator element in the parent. This continues recursively as long as this is necessary. At the last resort, this means the creation of a new root node. Since the height of the tree is logarithmic and the maximum permitted size of the nodes is constant, the total insertion is at worst logarithmic in time. In the case of deletion, there are several possibilities. The simplest case is to delete from a leaf that is large enough that deleting from it will not cause a problem. Then we simply remove the value and we are done. However, it may also be that we need to delete from the root, or from an internal node. In this case the task is more complicated, but it can be seen that the speed of the necessary steps is logarithmic. When searching, we proceed in a similar way to the Red -Black Trees, starting from the root, looking for the two keys between which the element we are looking for falls,

and proceeding towards the child between them. If we find the item we are looking for in one of the tips we are done, if we get to a leaf and there is none there, then there is no item in the tree.

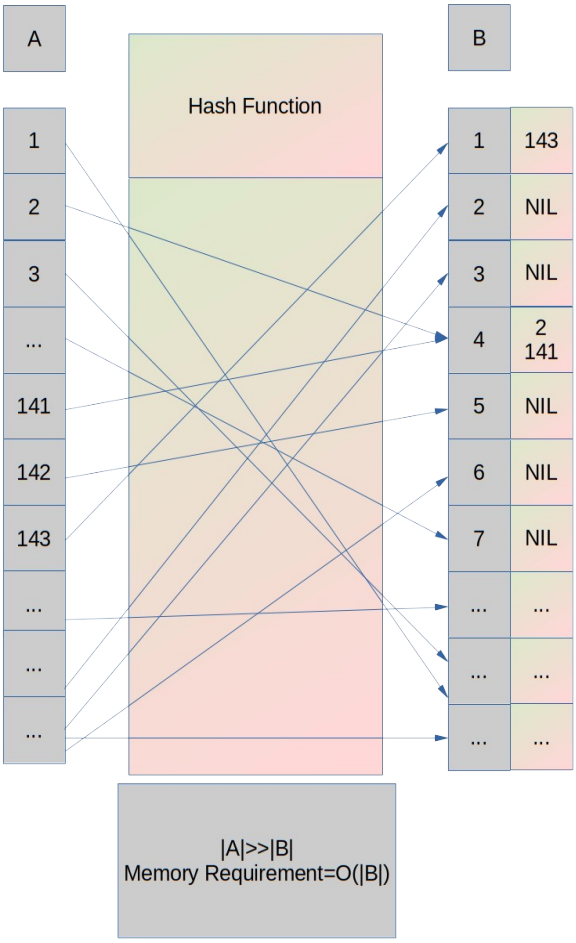
In addition to the default structure of B Trees, there are other variants. For example, the B+ Tree. The B+ Tree differs from the basic B Tree structure in, that the stored elements are located exclusively in the leaves. In this case, the higher levels are copies of the data stored in the leaves, such that they are the largest elements of the leftmost child. In addition, the leaves are also linked together as an unrolled linked list, thus speeding up sequential access. In addition, further variants of the tree B are known, which in practice improve the original structure in some way, but asymptotically all variants of the B Tree have the same speed.

1.5 Hash tables

A Hash Table is a search Data Structure that stores its elements in an array, usually as key-value pairs. The idea is that only a very small proportion of all possible keys will be used. For this reason, it is considered permissible to put several keys in the same place. To distribute the keys in the storage array, a function is needed which:

- runs in constant time (assuming that the size of the keys is bounded)
- deterministic, so it gives the same output for the same input
- uses all addresses in the storage array, close to uniformly
- does not give similar output for similar inputs.

The function that satisfies these is called a hash function, for which there are a large number of possible examples. When using the Hash table, the value of the key to be stored, deleted, or accessed is calculated using the hash function. We then perform the desired operation on the single value at the location pointed to by it as an index. This is the average and the best case.



6. Figure 6: Structure of the Hash Table

Unfortunately, since we need to assign several keys to an index, it is inevitable that different keys do use the same space from time to time. This is called a collision. For example, this is the case of location 4 in storage B in block B, shown in Figure 6, to which keys 2 and 141 have been assigned. There are several solutions to resolve these, such as using a linked list to store the elements, placed on the index, instead of direct storage. It can then be seen that the above operations (insert, delete, search) run in constant amortized time if a bidirectionally linked list is used and the size of the array is at most constant compared to the number of inserted elements.

The Hash Table usually uses Static Arrays, so no acceleration would result from Faster Mutation.

1.6 HAT

The Hashed Array Tree (HAT) contains nearly $\sqrt{n}$ ***elements*** of length $\sqrt{n}$, but rebuilds the whole data structure in linear time in case of mutation.

The structure is similar to the Accelerated Square Array, which will be described later. The advantage over conventional Arrays is that the additional memory usage is $\sqrt{n}$ whereas it is linear in conventional Dynamic Arrays. It consists of a Dictionary of size approximately $\sqrt{n}$, ***which*** stores the addresses of Dynamic Arrays. These Dynamic Arrays store the data themselves and are the same size as the Dictionary. The actual addresses per index *i are obtained by* dividing *i* by the size of the storage arrays. The quotient gives the index of the array of the element with the index we are looking for, and the remainder gives the index within it. Since remainder formation and division by integers is extremely slow, it is recommended to choose the Dictionary size of the first of the 2-powers whose square is greater than the number of elements in the storage array. So, for example, for 37 elements it is 8, since 64 is the smallest of the 2-powers greater than 37.

Then the division can be done by shifting the value to be divided to the right as many times as the number of powers of two (m), which for 8 is $\log_2(8)$, i.e. $m = 3$. Modulo is also much faster, since it can be done by taking only the smallest of the same power of 2 (m), leaving that many bits to the right, and setting the rest to 0. So, for example, for $m = 3$, Modulo by 8 can be done for any number in the 2 number system (for example 01010110_2) by leaving the 3 least significant bits of local value and setting the rest to zero(in the example this is 00000110_2).

During mutation, the whole data structure is rebuilt, and after determining the size of the Dictionary, the elements are stored in Vectors of the same size.

1.7 Comparison (with table analysis)

Asymptotic velocities describe the theoretical velocity at infinity. Therefore, the other extreme, i.e. the very small number of elements, can exhibit completely different properties, and the asymptotic properties only become significant as the number of elements increases. At low element counts, sub-operations with constant time requirements may become prominent, while at infinite limits they disappear completely. At very small sizes, Dynamic Arrays are usually the fastest for all tasks.

1. Table 1: Data structure Speeds

	Created by	Destruction	Mutation	Access	Search at
Dynamic Array (Vector, Deque, HAT)	**O(1)**	**O(1)***	**O(n)***	**O(1)**	**O(n)**
Chained List	**O(1)**	**O(n)**	**O(n)***	**O(n)***	**O(n)**
Fa	**O(1)**	**O(lg n)**	**O(lg n)**	**-**	**O(lg n)**
Cleaveboard	**O(1)**	**O(1)**	**O(1)***	**-**	**O(1)***
Accelerated Block	***O(1)***	**O*(sqrt n)***	**O*(sqrt n)***	***O(1)***	**O*(n)***

It is important to note that the Chained List only requires linear time if the index is to be reached. If the list has already been traversed anyway, in another operation, Insert and Reach require constant time. In addition, if the list is concatenated in 2 directions, or the previous index item has been noted, the delete is also constant time. In addition, an important factor is that these runtimes are amortized runtimes, however for the Hash table the worst case for Mutation and Search is O(*n)*. The destruction time of HAT is Θ(√***n***), and some Dynamic Arrays (Tiered Vectors) are capable of an amortized mutation time of O(√***n)***, even if this is not guaranteed in all cases.

You could also consider using Dynamic Arrays as a search data structure, by sorting the elements in a row. Since it then takes $O(\log_2 n)$ time to find a given element in a binary search, this is significantly better than searching in unordered Dynamic Arrays, but only as good as search trees. The problem with this is that the insert and delete operation is linear in time, whereas it was logarithmic in time for the trees.

In addition, different data structures have been developed to solve different problems. Accordingly, any comparison should take into account the problem for which we are looking for a solution. Because of the different uses, the Accelerated Array is best compared with Dynamic Arrays. In addition, occasionally, the characteristics of the physical device may also play a role in determining which is the best solution. In the case of possible new technologies, it is worth bearing in mind that most complex data structures are built from simpler data structures, so improving or replacing simpler data structures with an alternative

may also affect the speed of more complex data structures. In principle, there is no objective best among data structures, even in terms of asymptotic times. However, some hardware properties may create differences between properties that appear to be identical. Such is the case of the higher practical speed of B-trees compared to red black trees, since B-trees request more valuable data during a memory access, since keys are usually located sequentially in memory. However, in the actual implementation the issue becomes more complex and the possibility of a given improvement does not guarantee better speed. For example, if extremely large keys are used, up to the size of a cache line, this advantage of B-trees is lost.

2 Accelerated Array

An Accelerated Array is a data structure that stores its elements in an indexed manner, similar to a Static Array. Vector and Deque which are enhanced versions of Static Array that implement insert and delete operations in linear time.

The purpose of the Accelerated Array is to allow for a faster insertion and deletion time while maintaining a constant index access time. This is achieved by not storing the elements in one array continuously, but by spreading them over several smaller Dynamic Arrays of nearly equal length.

2.1 Basic Idea and Structure

The 3 components of the Accelerated Array:

- Metadata

  Metadata stores key data such as the current population, knowledge of which is essential for operation. In addition, any additional properties that we want to keep in temporary variables for speed or readability.
- Upper Array

  The Upper Block stores the addresses of the Lower Array.
- Lower Arrays

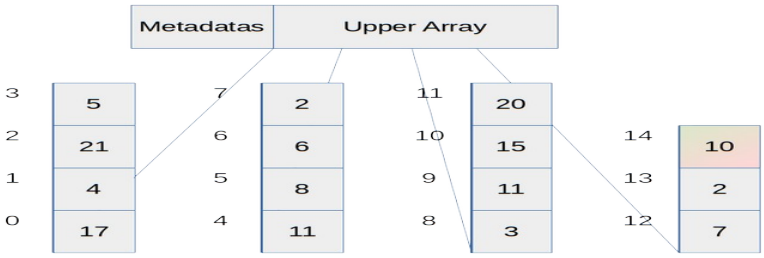
  The Lower Arrays store the data, these must be Deques, or another linear index based data structure that implements insertion and deletion operations at the ends in constant time.

The Accelerated Array avoids the large amount of data movement required when inserting data elements by storing the data in smaller vectors of length $\sqrt{n}$ or $\sqrt{(n)} + 1$ with a total count of $\sqrt{n}$ or $\sqrt{(n)} + 1$ for n elements instead of one large vector. The square root of the integer variable n is to be interpreted as the lower neighbouring integer.

1. The order of the data is determined, first by the location of the storage array, then by the position within the storage array. Lower Arrays are stored in the Upper Array, by address. Since the Accelerated Array reaches its elements by logical index, we can speak of a logical index, which is the position of an element in the sequence of stored elements. In addition, we can talk about a physical address, which here, is an integer

pair of numbers that indicates, first in which array an element is located, and then the index of the element within that array. Each logical address has a physical index, and vice versa.

An Accelerated Array is called square where each Lower Array is equal in size to the number of Lower Arrays.



7. Figure 7: Accelerated Array structure

2.2 Description and analysis

When an element is inserted in the Lower Array, moving data in proportion to the length of the Lower Block requires moving only √***n elements***. The Data Structure grows by first filling a „√(n)+1”th Lower Array from the bottom up, and then adding a new element to the end of each existing Lower Array. Since the logical order of the elements depends primarily on the containing Lower Array, it implies that the same element is stored in in the same logical place if it is at the end of the *xth* Lower Array or at the beginning of the *x* + 1th Lower Array.

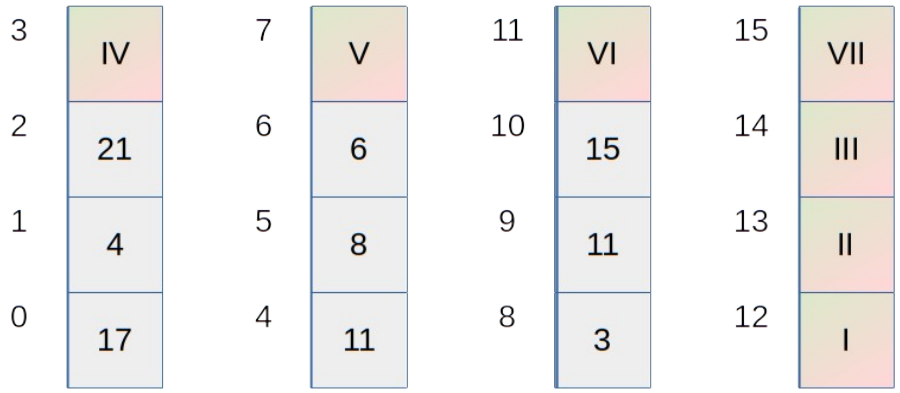
It follows that the index of an element removed from the end of a Lower Block and placed at the beginning of the following Lower Block does not change. The same also works in the opposite direction.

If a new element is placed at the beginning of a Lower Block and then the last element is removed from the end, its length does not change.

By taking advantage of these two facts, the physical distribution of the elements of the Lower Arrays can be changed without changing the logical order. This is possible because, if

in an arbitrary Lower Arrays an insertion happens, the surplus can be taken to the desired Lower Array without changing the logical order. This also works when deleting.

To keep the Lower Arrays close to $\sqrt{n}$, it is necessary to balance them. When rebalancing, the direction of the surplus is moved from the insertion's initial Lower Array to the direction of the next growth location.



8. Figure 8: Accelerated Array Growth

The Accelerated Array is empty when created. In this state, the insertion is trivially done into a new lower Array. The address of the new lower Array is placed in the Upper Array.

Since 1 is a square number, from here the array expansion can be described as a construction on a square core. Figure 8 shows the transition between a 9-element and a 16-element Accelerated Array. When expanding the square kernel, a new Lower Array is inserted first, in sequence after the others, which is indicated as column 4 in Figure 8. This is called the New Array. It is filled in at the same time as the existing Lower Array. This is the row of Roman 1,2,3 elements.

Each block is then increased by 1, in sequence, as shown in Roman 4.5.6. This results in a square structure one larger.

With the Deque operation, the excess is removed and placed in the next Lower Array, iterating until it is in the desired location. Since the blocks between the Insertion and the Growth Lower Block have all been increased in size once and decreased in size once, they are not resized.

2.2.1 Creation

In order to use a Data Structure, you must first be able to create it. In this case, this means an empty Data structure in which we perform some initializations. If we want to store a copy of another Dynamic Array in this way, we can then insert its elements in a row, as described later.

Creating the empy Structure:

I Creating the Upper Array

II Creating of the very first Lower Array

III create and populate a constant-size structure storing internal data, where it is necessary to store the population, but it is also capable of storing additional data, such as the rounded-down square root of the population, which gives the side length of the square kernel and how many extra elements are on top of that kernel.

Create():

1 upper_array ← vector()

2 insert(upper_array, deque())

3population ← 0

4popsqrt ← 0

5popextra ← 0

```
accarr() : metaData(0){
content = new deque<SQ<T> *>();
content->push_back(new SQ<T>());
}
```

Constant size memory reservation (at least 2, for the first Lower and Upper Array, but you may want to reserve more Lower Arrays in advance) by declaring exactly 4 variables with values, then modifying one of them in 1 step. Since the Create operation performs a constant number of operations, each of which is of constant time, the Create operation is also of constant time.

Θ(1)

2.2.2 Destruction

If necessary, we should also be able to destroy data structures that are no longer used, so that we don't run out of memory. Destroy all the Lower and then the Upper Array and the metadata.

Delete():

1 for each i element of top_array

2 do delete(*i*)

3 delete(upper_array)

```
~accarr(){
for (auto i : *content)
delete i;
delete content;
}
```

Free all the elements of the Upper Array (Lower Arrays), then the Upper Array itself, $\sqrt{n} + 2$ deletions in total, worst case. In the fastest case, there is only 1 Lower Array, so then 2 deletions are needed.

$\Theta = \sqrt{n}$

2.2.3 GetRelPos

To perform operations on any index, we need to know where it actually is, in terms of physical place. We can retrieve this with the GetRelPos function, which gets a valid logical index and returns a physical index.

How it works is as follows:

- At 0 (0,0) is the return value.
- If the New Block is not yet filled, the quotient of the non-last Lower Array length and the index *i* searched for, gives the Lower Array searched for, and the modulus gives the internal index within that array:

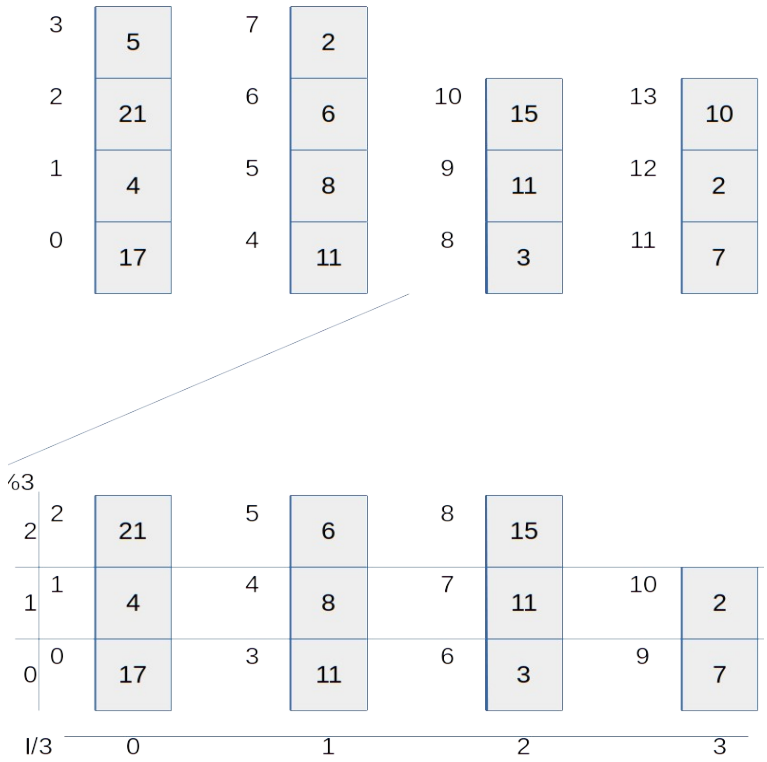
  $(i / \sqrt{n}, \sqrt{n} \mid i)$.
- If the New, last array is full, the size of the Lower Arrays is increased by 1 inorder. First, it must be determined whether the index being searched for, falls within the range of increased arrays.

  If so, then we proceed according to Point II, counting with $\sqrt{n} + 1$ long Lower Arrays.

If not, then subtract the number of elements stored in this way from i, calculate the index sought according to Point II, and then add, to the upper index calculated as in Point II, the number of Lower Arrays incremented.

In Figure 9, the index of elements is shown by the number to the left of them. Here you can see an Accelerated Array that is currently growing at the end. It can be seen that as the Lower Arrays are traversed in sequence, first the modulo value is filled up, then when it reaches full length, the quotient increases by one and the residual falls back to 0. Accordingly, the entire data structure is indexed throughout. An invalid physical address at the end of the last Lower Block can only be accessed by an invalid logical address.

The Accelerated Array growing on the Lower Array tops can be traced back to 2 different cases, all of which are manageable, based on the Accelerated Array growing at the end. At this point, the index you are looking for may fall within or beyond the increased length Lower Arrays. This is determined by dividing the index sought, by $\sqrt{n} + 1$. If the quotient is greater than or equal to the increased number of long Lower Blocks, then it falls outside the increased Lower Blocks. Since all elements stored outside the square core are stored either in the New Array or in extra members of the other arrays, the number of Incremented Lower Arrays is equal to the difference between the number of elements stored outside the square core and the number of elements stored in the New Array. When the $\sqrt{n} +$ 1th New Array is incremented, the next square core is reached.



9. Figure 9: Indexes of elements

GetRelPos(*index*):

```
1       if popsqrt = 0
2               then    return pair(0, 0)
3       if popextra <= popsqrt
4               then    return divmod(index, popsqrt)
5       else
6               if (popextra - popsqrt) > index / (popsqrt + 1)
7                       then    return divmod(index, popsqrt + 1)
8               else
9uninced ← index - (popextra - popsqrt) * (popsqrt + 1)
10uc ← divmod(uninced, popsqrt)
11                      return list(popextra - popsqrt + uc.first, uc.second)
```

```
pair<int, int> getRelPos(int index) const {
if (popsqrt == 0){
  return pair<int, int>(0, 0);
```

```
}
if (popextra < popsqrt){
  return divmod(index, popsqrt);
}
else{
        if ((popextra - popsqrt) > index / (popsqrt + 1)) /{
        return divmod(index, popsqrt + 1);
  }
  else{
        int uninced = index - (popextra - popsqrt) * (popsqrt + 1);
        auto uc = divmod(uninced, popsqrt);
        return pair<int, int>(popextra - popsqrt + uc.first, uc.second);
        }
  }
}
```

When the index is determined, the case where the population is 0 is treated first. Then, assuming a valid index has been retrieved, which can only be the case for insertion, this value is 0. This case needs to be treated separately because the physical index requires division, and modulus, by the square root of the population.

We then check whether the New Array is full. This can be easily obtained from the stored metadata without loading the New Array. To do this, we check if the number of elements stored outside the square core (Popextra) is greater than the number of elements that can be stored in the New Array.

If not, then the New Array is not yet full, and therefore a simple divide and modulo operation can be used to obtain the desired physical index, as shown in line 5. The index sought divided by popsqrt gives the index of the Lower Array and modulo of the same gives the index within it.

If so, then there are more extra elements stored than what can fit in the New Array, these are stored as extra members of the Lower Arrays at the ends and need to be corrected accordingly by the extra size of the first few Lower Arrays. To do this, we need the number of elements in the unincremented part reached by the index. This is extracted in the variable uninced in line 10. To do this, we need to subtract the number of elements falling into the incremented Lower Arrays from the index we are looking for. The number of elements in the incremented part is the length of the existing Lower Arrays(popsqrt + 1) multiplied by the number of incremented Lower Blocks (popextra - popsqrt). Then, the indexed value in the unincremented part is calculated as if it were the case in row 5. This gives the number of Lower Arrays after the incremented part and the index within it, in row 11. Finally, this is

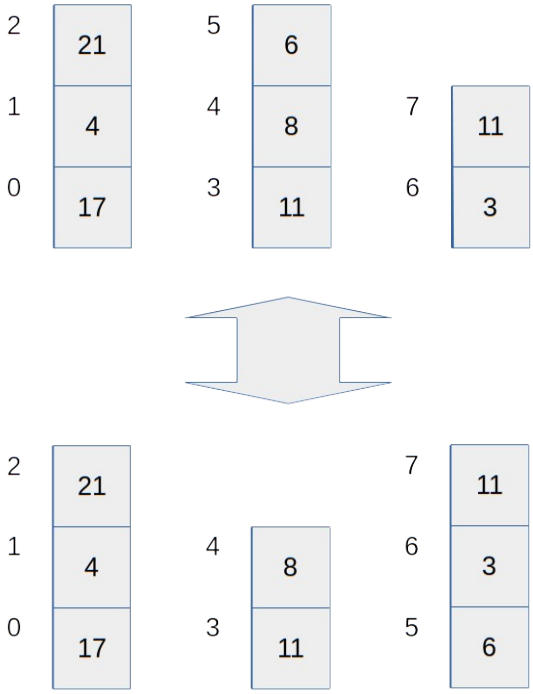
corrected by adding the number of incremented Arrays to the index of the Lower Array to be returned.

Since in all cases constant-time operations are required, without recursion or iteration, the total function itself is **Θ(1)**.

2.2.4 BalanceShift:

Balancing and its running time is of elementary importance in the Accelerated Array. Mutation then rebalancing is the idea behind Red Black Trees and many other balanced data structures. In the case of Accelerated Array, balancing takes advantage of thhe property of Deques that doublestack operations all run in constant amortized time (not including larger storage rewrites).

The logical index of an element transferred from the end of a given Lower Array to the following beginning does not change. By the same reasoning, the logical index of an element moved from the beginning of a given Lower Array to the end of the preceding Lower Array does not change either. Even if an element is inserted into or removed from a Lower Array, the length does not change. The balancing is made possible by the fact that, because of the facts mentioned above, the logical index and the physical index can be treated separately. Thanks to this, after mutation, the structure of the Accelerated Array can be restored.

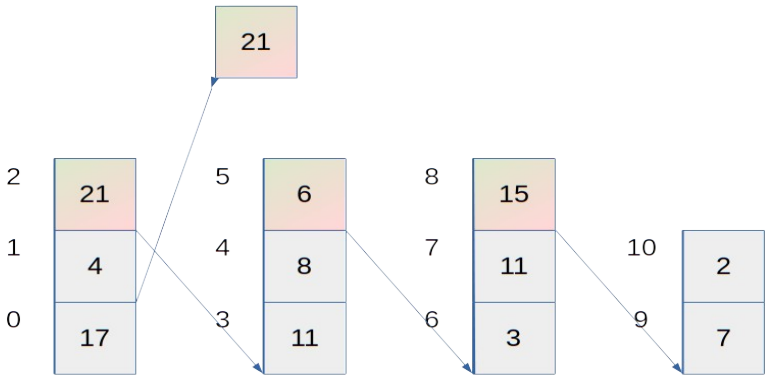


10. Figure 10: Balance shift illustration

Figure 10 shows two Accelerated Arrays with the same logical structure but different physical structures. The lower one shows a state that occurs after deletion from the middle Lower Array, and the upper one shows a corrected state of the latter. After the deletion, the logical order is not violated, only the distribution of the elements. In case of an upset of the balance, to arbitrarily move the balance, it is enough to go from the disturbance source Lower Array to the Target Lower Array, whose index in the Upper Array is given in the parameters. To remove a disturbance at one Lower Array and place it at the correct Lower Array of the following Lower Array, we carry it from Lower Array to Lower Array, by pushing and popping at the ends, without changing the logical order. The choice of the appropriate parameters for balanceShift is the responsibility of the function that calls it.

To do this, we look at the direction in which we want to move. If we move from a Lower Array with a lower index towards a Lower Array with a higher index, we take the elements from the end of the Lower Array and place them at the beginning of the next Lower

Array. If from a higher index to a lower index, the elements are taken from the beginning of the Lower Array and placed at the end of the next Lower Array. If the starting and arriving Lower Array are the same, no action is required as the equilibrium is not upset, the mutation has occurred at the intended location of the change.



11. Figure 11: Offsetting surplus

Figure 11 shows the insertion of the value "21" after the index 0. With this, an insertion occurs in the first Lower Array, which causes an extra element to be added here. To solve this, Insert, like Delete, calls BalanceShift. In this case, this is a function call where the shift of the excess is done from the first to the last element. Then the Array with value 21 (which was originally at logical index 2) is taken off Lower Array 1 and moved to the beginning of Lower Array 2. Then its logical location is not changed, but the physical surplus is moved closer to the desired location. Then the last element of Lower Array 2, "6", is poppe d back and placed at the beginning of Lower Array 3. Finally, the "15" at the end of Lower Array 3 is moved to the beginning of the fourth Lower Array. This corrects the structure.

```
BalanceShift (from,towards):
 1      if from < towards
 2            then    temp ← pop_back(upper_block[from])
 3                    fori ← from + 1 to towards
 4                          do      push_front(top_block[i])
5temp ← pop_back(upper_block[i])
6temp ← push_front(upper_block[from])
 7      else iffrom > towards
 8            then    temp ← pop_front(top_block[from])
```

9 for*i* ← *from* - 1 back_to towards

10 do push_back(top_block[*i*])

11temp ← pop_front(top_block[*i*])

12temp ← push_back(upper_block[*from*])

```
void balanceShift(int from, int to)
{
if (from < to){
  T temp = ((*content)[from])->pop_back();
  for (int i = from + 1; i < to; i++){
        temp = ((*content)[i])->revplace(temp);
  }
  ((*content)[to])->push_front(temp);
  }
else if (from > to){
  T temp = ((*content)[from])->pop_front();
  for (int i = from - 1; i > to; i--){
        temp = ((*content)[i])->placing(temp);
  }
  ((*content)[to])->push_back(temp);
  }
}
```

In the actual code, we examine the relation between the source and destination index. If the source index is smaller than the destination index, we take the elements from the ends and place them at the beginning of the next one. To do this, we first remove an element from the end of the starting Array, and then we go through all the Lower Arrays between the starting Array and the destination Array, inserting one element at the beginning and removing one element at the end. Finally, the last element is inserted at the beginning of the destination Array. If the Target is larger than the Starting Index, we do the same, except that we take the elements at the beginning of the Lower Blocks and insert them at the end of the Lower Blocks.

This shows that during BalanceShift we perform a constant step outside of the iterations or at most one iteration, as shown in lines 3 and 10. This is the distance between the source Index + 1 and the destination Index, i.e. the absolute value of their difference. The source and destination indices are both, within the Upper Array . So their difference is at most the length of the Upper Array. This is O(√n), which is not changed by the weaker constant + 1 factor. The cycle kernel takes exactly 2 constant time steps. All this gives a speed of O(√***n*** * 1) = O(√***n***). The mean line segment gives an expected speed of Θ(√***n*** / 3) =

Θ(√*n*). In the best case, we do not enter the iteration, in which case the running time is constant.

2.2.5 Additional auxiliary functions

divmod is an internally used function. Its job is to simply perform the divide and modulate operations with the same values.

To do this, simply return with the list of the quotient and the modulo.

divmod(*a*,*b*):

1 return list(floor(*a* / *b*),*a* % *b*)

```
pair<int, int> divmod(int a, int b){
return pair<int, int>(a / b, a % b);
}
```

Since each step is constant time, the total function **Θ(1)**.

Population growth implements the updating of metadata after insertion. In addition, it determines whether a new Lower Array needs to be inserted.

To do this, it remembers the old square root of the population, then updates the population value, its square root, and the number of elements stored beyond it. Finally, it returns whether the new square root is not the same as the old one (which happens if and only if a new square has been reached).

incPop():

1population ← *population* + 1

2oldSQRT ← *popSQRT*

3popsqrt ← *floor*(sqrt(*population*))

4popextra ← *population* - *popsqrt* * *popsqrt*

5 return*oldSQRT* = *popSQRT*

```
bool incPop(){
population++;
auto oldsqrt = popsqrt;
popsqrt = SQRT(population);
popextra = population - popsqrt * popsqrt;
return oldsqrt != popsqrt;
}
```

Population reduction updates metadata after deletion. In addition, it returns whether to delete the last Lower Array.

To do this, similar to population growth, it remembers the old square root of the population, then updates the population value, its square root, and the number of elements stored beyond it. Finally, it returns whether the new square root is the not same as the old one (which happens if and only if a new square is reached).

decPop():

1population ← *population* - 1

2oldSQRT ← *popSQRT*

3popsqrt ← *floor*(sqrt(*population*))

4popextra ← *population* - *popsqrt* * *popsqrt*

5 return*oldSQRT* = *popSQRT* and *population* > 0

```
bool decPop(){
population--;
auto oldsqrt = popsqrt;
popsqrt = SQRT(population);
popextra = population - popsqrt * popsqrt;
return oldsqrt != popsqrt;
}
```

Since each step is constant time, the total function time is **Θ(1)**.

The calcInsertPlace returns the location of the Increment. For a perfect square container, it starts a new Lower Array, fills it up to the same height as the others, and then increments each vector from the beginning by one element.

calcInsertPlace():

1 if*popextra* < *popsqrt*

2 then return list(*popsqrt*, *popextra*)

3else

4 then return list(*popextra* - *popsqrt*, *popsqrt*)

```
pair<int, int> calcInsertPlace() const{
if (popextra < popsqrt){
  return pair<int, int>(popsqrt, popextra);
  }
else{
  return pair<int, int>(popextra - popsqrt, popsqrt);
  }
}
```

Since each step is constant time, the total function is **Θ(1)**.

The calcDeletePlace returns the location of the correct reduction place during the deletion. This function can be used to retrieve, for a deleted element, from where the new

balancing element should be brougt in its place. In a sense, it is the opposite of calcInsertPlace.

calcDeletePlace():

```
1       ifpopextra == 0
2               then    return list(popsqrt - 1, popextra - 1)
3else   if popextra <= popsqrt
4               then    return list(popsqrt, popextra)
5       else
6               then    return list(popextra - popsqrt - 1, popsqrt)
```

```
pair<int, int> calcDeletePlace() const{
if (popextra == 0){
  return pair<int, int>(popsqrt - 1, popsqrt - 1);
  }
else if (popextra <= popsqrt){
  return pair<int, int>(popsqrt, popextra);
}
else{
  return pair<int, int>(popextra - popsqrt - 1, popsqrt);
  }
}
```

Since each step is constant time, the total function is **Θ(1)**.

2.2.6 Mutation

For the logical index given **for insertion**, we first find the actual physical location in the data structure, which is the corresponding Upper Array and the index within it. This location is then inserted in the Lower Array, and the Balansceshift operation shifts the excess element towards the growth/shrinkage point. After this, the internal data of the Accelerated Array is recalculated in constant time.

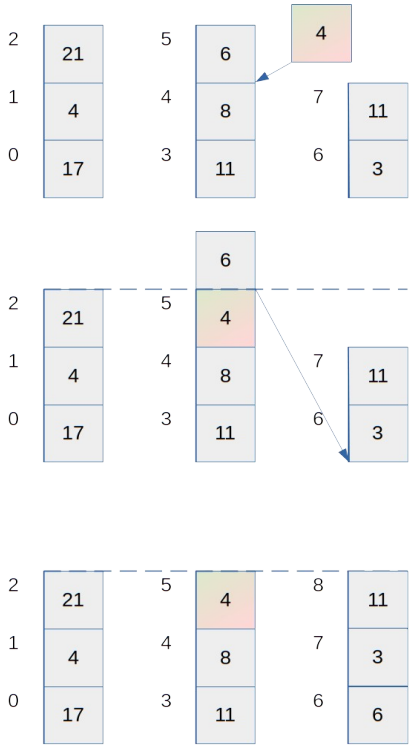
Insert(*index*, *value*):

```
1       place ← getRelPos(index)
2       insert(upper_block[first(place)], upper_block[second(place)], value)
3       balanceShift(first(place), first( getInsertPlace( population)))
4todo ← incPop()
5       if todo
6               then    push_back(upper_block, vector() )
```

```
void insert(int index, const T &value){
```

```
auto to = metaData.getRelPos(index);
((*content)[to.first])->addTo(to.second, value);
balanceShift(to.first, metaData.calcInsertPlace().first);
auto todo = metaData.incPop();
if (todo)
content->push_back(new SQ<T>());
}
```

Figure 12 shows an example of the entire process of Insertion. After GetRelPos finds the desired physical location, which here is after index 4, into the Lower Array, the insertion is performed. At this point the inserted value "4" is in a good logical location but there is a surplus in the second Lower Array where the insertion was made. Then, at the growth location specified by CalcInsertPlace, the balanceShift shifts the element surplus, thus eliminating the incorrect distribution of elements. If the insertion would have been in a location, logically, where the physical increment falls, balanceShift does not perform any operation. After all of this, the Accelerated Array Metadata is updated and the update function returns whether it is necessary to add a new Lower Array to the Upper Array. This is the case in this instance, as a new square has been reached by the element structure(3×3).



12. Figure 12: Insertion procedure

The insertion itself does not involve iteration or recursion, so the functions called decide its speed. There are two called functions that are not constants, into Lower Array insertion at arbitrary locations and BalanceShift. In average and worst case, the Lower Array insertion runs in $\sqrt{n}$ time, since the Lower Array is nearly $\sqrt{n}$ long (either exactly that or $\sqrt{n} + 1$). In the best case, we insert at the end which allows the insertion time to be constant. The BalanceShift is at best unnecessary, but the odds of this are only 1 in $\sqrt{n}$ (the probability of falling into the one to be incremented out of all the Lower Array). Otherwise, there are 2 possibilities, that the data structure is incremented at the end, or, that it is evenly distributed in some arbitrary Lower Array with equal chance. Each has a 50% chance. The first is length / 3, which here is $\sqrt{n}$ / 3, the other is average distance from endpoint, which is length / 2, in this case $\sqrt{n}$ / 2, so on average 5 / 12-ed $\sqrt{n}$.

Since, in balancing, one hand of the insertion/erasure time is proportional to the length of the Lower Array and on the other hand, is proportional to the length of the Upper Array, it

is minimal, if the two are equal, since the sum of the adjacent sides of rectangles of the same area is minimal if the reactangle is a square.

In the best case, the desired logical index insertion falls in the growth position, so there is no further non-constant-time action to take, as Balanceshift does nothing. Furthermore, the growth point in preferable case falls at the end of a Lower Array.

The runtime is **$\Theta(\sqrt{n})$.**

During **Deletion**, similar to Insertion, the actual location is found with GetRelPos, the deletion is done, and then the balancing is done from the decrease location with BalanceShift. The internal data is then recalculated.

Delete(*index*):

1place ← *getRelPos*(*index*)

2 erase(upper_block[first(*place*)], *upper_block*[second(*place*)])

3 balanceShift(first(getDeletePlace()),first(*place*))

4todo ← decPop()

5 if *todo*

6 then pop_back(*top_block*)

```
void erase(int index){
auto to = metaData.getRelPos(index);
((*content)[to.first])->deleteFrom(to.second);
balanceShift(metaData.calcDeletePlace().first, to.first);
auto todo = metaData.decPop();
if (todo)
content->pop_back();
}
```

Since there is no iteration or recursion, the called functions decide the runtime here too. Among these, in Lower Array Delete and BalanceShift are the non-constant-time ones. The Lower Array element deletion behaves similar to the Lower Array element insertion in terms of time, i.e. it is constant at best and √n at worst and average. All this means that the running time during the Deletion is here also, Best case Constant, worst and average case **$\Theta(\sqrt{n})$.**

2.2.7 Access

When **writing to** a given location, we need the physical location of the destination index, which tells us which Lower Array to access and which element within it. For the physical address, we simply call the GetRelPos function with the target logical index. The first

element of the return value is the index of the lower array we are looking for, the second value is the index within it. We then access this element and perform the write operation there.

Writing(*index*, *value*):

1tarvector ← *first*(getRelPos(*index*))

2place ← *second*(getRelPos(*index*))

3tarvector[*location*] = *value*

```
void setAt(int index, const T &value) const{
refAt(index) = value;
}
```

Since each step is constant time and there are fixed chunks of them, the whole operation is constant time. **Θ(1)**.

When reading from a given index, we proceed in a similar way to writing. We find the Physical Address associated with the resulting index, and then read the value indicated by the second element of the Physical Address from the Lower Array indicated by the first element of the Physical Address.

Reading(*index*):

1tarvector ← *first*(getRelPos(*index*))

2place ← *second*(getRelPos(*index*))

3 return*tarvector*[*location*]

```
T getAt(int index) const{
return refAt(index);
}
```

Since this requires a constant number of constant-time sub-operations, the running time is **Θ(1)**.

Code implementing reference operations refers to the following reference call function, which is a sequence of constant-time suboperations, also **Θ(1)**:

```
T &refAt(int index){
auto pos = metaData.getRelPos(index);
return ((*content)[pos.first])->access(pos.second);
}
```

2.2.8 Search at

Even though the Accelerated Array is not intended to be a Search data structure, you may still want to search for items in it. This might occur, for example, if we store the contents of the nodes of B Trees in it. In this case, similar to the Dynamic Arrays currently used for this purpose, it is useful to sort the contained elements in a sequence. It is important to mention here that, especially for small numbers of elements, Insertion Sorting can be promising, as it is one of the few sorting algorithms that de facto uses insertion and deletion, making it faster than the one used for Vectors and Deques. A binary search can then be performed on the sorted elements. This has a runtime of **$O(\log_2 n)$.** The pseudocodes presented here simply return the value, or logical index.

```
Search_ordered( value):
1bottom_railing ←
2upper_limit ← population - 1
3       while lower_limit < upper_limit
4               do      middle ←        floor((lower_limit + upper_limit) / 2)
5                       if      value = Read(middle)
6                               then return middle
7                       else if value < Read(middle)
8                               then upper_limit ← middle - 1
9                       else
10                              then    lower_limit ← middle + 1
```

You may also want to search on unordered items. In this case, we simply start going through all the elements and stop the search when we find the element we are looking for. Since, in the worst case, we need to examine all elements to see that the element we are looking for is not in the data structure, the runtime for this is **$O(n)$.**

```
Search_Unordered( value):
1       for i ← 0 to population - 1
2               if Read(i) = value
3                       then    return value
4               else
5                       then    return false
```

2.2.9 Rotation

The Insertion Sort consists of a succession of insertions and deletions. This corresponds to rotating a sub-block of an Array between given indices. In order to perform an efficient Insertion sort, instead of naively performing a Delete and Insertion one after the other, it is recommended to use a merged algorithm that performs the balancing between the Lower Block of Insertion and Deletion. Then the operation needs nothing more than the 2 two indices from which the desired value is deleted, and then to which it is inserted. These two indices are essentially the first and last elements of the rotated subarray. The algorithm implemented here performs a single rotation, deleting from the second index specified as the variable and inserting in front of the first.

Rotation (*first*, *last*):

1insertion point ← getRelPos (*first*)

2 *delete* location ← getRelPos (*last*)

3 *value* ← Read(*last*)

4erase (*top_block*[first(*delete_place*)], *top_block*[second(*delete_place*)])

5 insert(top_block[first(insertion point)],top_block[second(insertion point)],temp)

6 balanceShift(first(insertion point), first(deletion point))

Since this function is constant and *a sequence of functions with time* $\Theta(\sqrt{n})$*, it is itself* $\boldsymbol{\Theta(\sqrt{n})}$**.**

3 Implementation, measurements

I have created an implementation for the Accelerated Array. There were several reasons for this, first of all I wanted to make sure that it actually did its job and that there were no unnoticed bugs, especially in the edge cases. I then used the implementation to compare speed against C++'s built-in std::vector. I then analysed the measurements. It is important to point out that for the sake of readability, I did not perform any major optimizations, trying to stay as close to the pseudocode as possible. However, I used several local variables that are queried at index access but only change during mutation. One example is a boolean variable called endLoad, which stores whether the Accelerated Array, in the new vector, is incremented(true), or at the end of the vectors(false). These were essentially just handwritten output caches for some internal built-in functions.

3.1 Implementation

The implementation was done in C++. There were several reasons for this. As one of the best known programming languages, the ready-made implementation written in it has a very wide audience. It is sufficiently low-level to allow almost all the necessary steps to be covered. It supports object orientation and templates, which makes it easy to test and use.

I have defined a separate class for the Metadata and a separate class for the Lower Arrays. The Upper Array, on the other hand, since it does not have a separate scope, I simply implemented as a Dynamic Array storing the Lower Arrays. During the implementation, I created a template class that stores the Metadata and the Upper Array as internal data members. I have structured the whole project in this way to be able to develop the system element by element. Since I knew the requirements before starting the implementation, I was able to test component by component as soon as I finished a component. Once the components were complete, I tested the whole system.

3.2 Verification

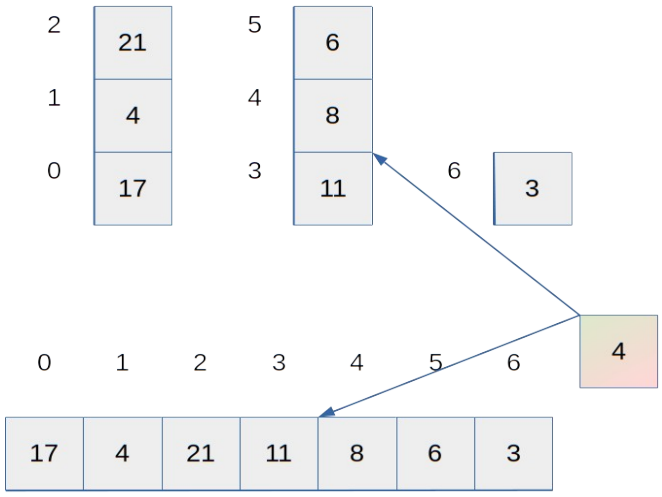
The primary purpose of the testing was to make sure that the implementation actually did what it was supposed to do. This is called verification. Verification was also necessary

because without it it is meaningless to measure the performance of a program, since it doesn't matter how fast a bad program runs. In essence, verification was to make sure that my Accelerated Array implementation properly implements the operations and behavior expected from a Dynamic Array.

The Dynamic Array must allow insertion and deletion operations for a given index, and must be able to retrieve and modify a given value on an index basis, in constant time. In addition, after inserting a value *x into a* given index i, the data structure must return *x* when requesting element *i*. In addition, elements with a larger index than this shall all be shifted by one index towards a larger index value. Similarly, on deletion, elements with an index greater than the index *i* deleted shall be moved to an index one less than the index *i* deleted. In addition, after overwriting a given value at index i, index *i shall* store the new value until something else justifies a change.

An important feature of the item-by-item check for several functions was that it was possible to formally determine in advance, in certain cases, what the correct return value was. Accordingly, after calling it, I could easily check that it returned the correct value. Once the elements were individually correct, I had to test the entire data structure. Correct operation of all the elements alone is not sufficient, since possible semantic errors will only be revealed when the whole system is running, and errors in the way the parts are connected together, in either half of the connection, will not be detected when tested separately.

In developing the Full Data Structure check, I have assumed that the built-in std::vector in C++ works according to the Dynamic Array properties. Based on this, I performed the same operations on both an instance of std::vector and an instance of Accelerated Array. I generated the operations based on random brute force, and also used a manually generated sequence of instructions to check the edge-cases.



13. Figure 13: Insertion of 4 after index 3

The operation sequences included insertion and deletion, with valid indexes, and write operations. At the end of the tasks, the two data structures were read out in sequence and compared element by element. Figure 13 shows the tested Accelerated Array Example and the control Vector. In the Figure, both store the same values.

The tests also revealed several minor implementation bugs that were not found in the component-by-component testing. These were corrected after successful testing and the implementation was considered complete.

3.3 Measurement

To perform the measurements, I used the implementation written in the C++ programming language, also used for verification. The implementation was based on pseudocodes without any major optimizations, since the main goal was validation. I did not use any of the additional optimizations mentioned in the Summary.

I performed the same measurements on a C++ built-in library std::vector and on the Accelerated Array. The measurements were performed on 20-5120 elements, with multiple iterations. Smaller waits were inserted between measurements to minimize the impact of the measurements on each other, however, all cases (element and operation ratio combination with elements of a given size) were measured in one run, so this could bias the final result for large element counts. The measurements also varied the ratio of mutations to hits, from 1:1 to

1:256, in favour of mutations. The measured values are the ratio of the run times with the Accelerated Array time in the counter.

2. Table 2: 32bit float storage speed

element /ratio	1	2	4	8	16	32	64	128	256
20	*15.77*	*5.38*	*12.95*	*14.41*	*11.89*	*15.83*	*12.22*	*15.83*	*11.82*
40	*11.37*	*8.27*	*12.52*	*12.64*	*12.60*	*10.79*	*9.91*	*8.00*	*11.84*
80	*9.36*	*5.10*	*8.92*	*5.83*	*7.41*	*8.05*	*7.30*	*7.31*	*6.70*
160	*6.09*	*4.43*	*4.83*	*4.80*	*5.37*	*4.01*	*5.47*	*4.04*	*5.39*
320	*4.61*	*3.99*	*4.49*	*3.59*	*4.28*	*4.12*	*2.45*	*2.93*	*2.88*
640	*3.10*	*2.18*	*1.74*	*1.23*	*1.86*	*1.81*	*1.63*	*1.60*	*1.88*
1280	*1.20*	*1.31*	*1.55*	*1.53*	*1.32*	*1.70*	*0.92*	*1.31*	*1.42*
2560	*1.30*	*1.17*	*1.16*	*1.13*	*1.02*	*1.24*	*1.02*	*0.99*	*1.15*
5120	*0.94*	*0.85*	*0.85*	*0.78*	*0.84*	*0.83*	*0.58*	*0.78*	*0.75*

3. Table 3: Storage of 8192byte element storage speed

	1	2	4	8	16	32	64	128	256
20	*1.066*	*0.684*	*1.063*	*1.397*	*1.728*	*1.878*	*1.396*	*1.644*	*1.222*
40	*1.040*	*0.720*	*1.155*	*1.091*	*1.024*	*0.852*	*0.653*	*0.487*	*0.718*
80	*0.580*	*0.359*	*0.554*	*0.328*	*0.443*	*0.486*	*0.482*	*0.510*	*0.425*
160	*0.381*	*0.292*	*0.318*	*0.353*	*0.365*	*0.244*	*0.324*	*0.215*	*0.235*
320	*0.176*	*0.167*	*0.145*	*0.100*	*0.117*	*0.111*	*0.077*	*0.077*	*0.072*
640	*0.101*	*0.074*	*0.061*	*0.048*	*0.056*	*0.053*	*0.050*	*0.047*	*0.048*
1280	*0.057*	*0.044*	*0.038*	*0.034*	*0.031*	*0.035*	*0.027*	*0.030*	*0.031*
2560	*0.037*	*0.029*	*0.025*	*0.024*	*0.023*	*0.026*	*0.024*	*0.023*	*0.025*
5120	*0.027*	*0.021*	*0.019*	*0.017*	*0.017*	*0.017*	*0.015*	*0.017*	*0.017*

The measurements show that the Accelerated Array performs better and better as the number or size of elements increases and the mutation rate increases. At best, the acceleration is more than 65-fold. From the measured data, it can be observed that the number of elements is more important than the ratio of operation types, since a much more drastic acceleration is

seen when stepping row by row than column by column for the same multipliers. The size of the elements is also crucial, with larger elements, the advantages of the Accelerated Array are more pronounced. Approached from the other direction, for very small element size and number, the larger constant coefficients of the Accelerated Array become dominant. These larger coefficients can be attributed to the GetRelPos inner function's divide and modulo operation, since both divide and modulo is extremely slow on modern x64 processors, compared to the multiply and add operations.

3.4 Analysis, usability

Even without further optimization, the Accelerated Array can be used directly where Vectors and Deques have been the fastest alternatives, even with insertion times, provided that they do not exploit the potential for rapid endpoint mutation of deques and vectors (unfortunately, the Accelerated Array itself is does that, in the Lower Arrays, so it is not worth embedding it in itself). Such a task is a datastore where the number of elements is not very high and the mutation rate is negligible compared to the reaches, but the large element sizes still make it a speedup to use. The Accelerated Array cannot provide a speed advantage over the Static Array where no mutation occurs.

Indirectly, they can be used as internal repositories in B trees, especially in leaves, where the mutation rate is the highest possible compared to the accesses. If the maximum number of elements in a node is *c*, *then* the mutation in the node for Dynamic Arrays is $\Theta(c)$, while the same is $\Theta(\sqrt{\boldsymbol{c}})$. This takes $c / \sqrt{\boldsymbol{c}} = 1 / \sqrt{\boldsymbol{c}}$ time, i.e., it results in an acceleration of $\sqrt{\boldsymbol{c}}$.

In addition, it can also accelerate algorithms, if they have been slowed down by mutating in Vector, or Deque. One of these is the Insertion Sort, which performs *n* mutations of size n/2 on average, so that it has a runtime of $O(n^2)$ using these data structures. This can be improved to $O(\sqrt{\boldsymbol{n}} * n)$ using an Accelerated Array. In addition to the existing algorithms, new algorithms may emerge that have been objectively too slow until now, due to the slow linear time of mutation in Dynamic Arrays. Since modern hybrid sorting algorithms often switch to $O(n^2)$ sorting algorithms, such as Insertion Sorting for recursive calls with small element counts, it is also viable that these may experience speedups due to the use of Accelerated Array.

3.5 Use of implementation

Although the implementation is mainly designed for verification and validation, it may still be used by someone else. In this case it is necessary to import the source code files into the project. It can then be referred to as a filled template when declaring variables. Because of the close syntax to the Dynamic Arrays in the C++ Standard (std::vector,std::deque), it is relatively easy to replace them. This requires replacing both the specific type names and the analogous or near-analogous functions. For example, replacing element access by iterators with direct index-based access. Indexing by square brackets is implemented in the Accelerated Array.

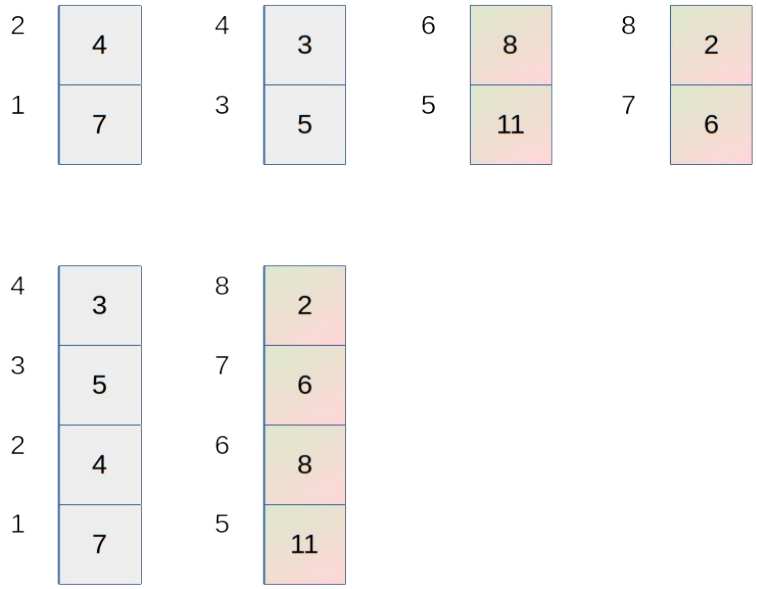
```
#include <vector>

vector<int> v;                  // empty
v.insert(v.begin(), 0);         // 0
v.insert(v.begin() + 1, 3);     // 0 3
v[1] = 2;                       // 0 2
v.erase(v.begin() + 0);         // 2
cout << v.at(0) << endl;
#include "accarr.h"
accarr<int> a;                  // empty
a.insert(0, 0);                 // 0
a.insert(1, 3);                 // 0 3
a[1] = 2;                       // 0 2
a.erase(0);                     // 2
cout << a.refAt(0) << endl;
```

4 Summary and further options

From all of these, it can be seen that the Accelerated Array may be able to provide faster alternatives for certain tasks, compared to known data structures, or it may be a starting point for possible new data structures. In conclusion, I would like to mention some possible and also impossible ways of further development, with a focus on their possibilities and problems. During the development of the Accelerated Array, I did not find any further possibilities for improvement in all aspects, but I did find some better ways of further development from certain points of view, listed among other things, in the Comments.

4.1 comments

- Because of the prefetch and cache capabilities of modern processors, 1-time indirection of memory addresses does not cause significant slowdown.
- Since the mutation at the end of the inner deques is constant time, but in the Accelerated Array this is not the case half of the time, so self-embedding does not involve acceleration.
- Beyond the naive implementation, one might consider choosing the size of the top vector as one of the two powers that estimate the root of the size from the bottom or the top. This would be advantageous because the divide and modulate operation can be replaced by bit shift and bit mask operations, thus significantly reducing the constant coefficient of index access, similar to HAT. Unfortunately, in this case, the worst case of mutation is O(n), since at least half of the elements must be moved, in the worst case, as shown in Figure 14, all of them, into a larger reallocated Deq. This would imply a large instantaneous overhead for large data sets. In the worst case, the coincidence of several such overloads could stall the operation of a database. The amortized case is still O($\sqrt{\boldsymbol{n}}$), and the constant time of access operations is, in practice, significantly improved, thanks to the replacement of slow modulo and integer divide operations by bit operations. The data structure described here, the Tiered Vector.



14.

Figure 14: Linear time rescaling

- You may want to centre the new growing Lower Array in the upper Array, halving the expected balancing time, half the time in half the cases.Half the balancing time is balancing the lower Arrays. When filling the New Lower Array, the new Lower Array will get the elements, if you balance towards one end the average distance is √n/2, if towards the middle, the average distance is √n/4. This complicates the getRelPos function
- a circular queue doublestack for the Lower Array, which allows push and pop operations to be performed with one memory access, instead of the 2 required for dequeues. This would further slow down the calculation of the correct index at access.
- GetRelPos can be greatly accelerated with polymorphism or function pointers, which are changed only when resized, otherwise just called, so constant path coefficients can be further reduced .
- Replacing integer division with float or fixpoint recirocal multiplication can also achive speedup. In this case precision and accuracy must be carefully considered.

bibliography

[1] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein, New Algorithms, Scolar Publishers, 1999.

[2] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein, New Algorithms, Scolar Publishers, 1990.

[3] Michael T. Goodrich, Roberto Tamaissa,Algorithm Design, Wiley, 1999.

[4] Edward Sitarski,https://www.drdobbs.com/database/algorithm-alley/184409965?pgno=5[2023.12.04.], 1996

[5] source: https://www.cambridge.org/core/journals/bulletin-of-the-australian-mathematical-society/article/average-distance-between-two-points/F182A617B5EC6DB5AD31042A4BDF83AE[2023.12.04.], 2009

[6] source: https://owlcation.com/stem/Which-rectangle-gives-the-biggest-area[2023.12.04.], 2022

[7] Michael T. Goodrich, Roberto Tamaissa,Algorithm Design, 1999.

Additional files

accarr.h

accarr_innerData.h

comparee.h

main.cpp

speedcomparison.h

SQ.h

tests.h

Table of figures

Table List