UNIVERSITY OF TARTU
Institute of Computer Science
Informatics Curriculum

Nikolai Voitsehhovski

Input Generation for Hashtable Algorithms

Bachelor's Thesis (6 ECTS)

Supervisor: Ahti Põder, Ph.D.

Tartu 2022

Input Generation for Hashtable Algorithms

Abstract:

The goal of this Bachelor's thesis was to create software solution that is able to generate input arrays for classical hashtable algorithms used in „Algorithms and data structures" course. Automated solution decreases amount of routine job for teacher and helps practising algorithms for students. In the process a Java application with minimalistic graphical user interface was implemeted, which allows generating input arrays for four different algorithms.

Keywords:

Algorithms and data structures, hashtable, Java, educational software

CERCS: P175 Informaatika, süsteemiteooria

Paisktabelialgoritmide sisendite genereerimine

Lühikokkuvõte:

Bakalaureusetöö eesmärgiks oli luua tarkvara lahendus aines „Algoritmid ja andmestruktuurid" käsitlevate klassikaliste paisktabelialgoritmide sisendite genereerimiseks. Töö käigus valmis minimalistik graafilise kasutajaliidesega Java programm, mida suudab neljale algoritmile genereerida etteantud parameetrite alusel sisendeid.

Võtmesõnad:

Algoritmid ja andmestruktuurid, paisktabel, Java, õpitarkvara

CERCS: P175 Informatics, systems theory

Table of Contents

Introduction 4
1 Solution Overview 5
1.1 Thesis Goal 5
1.2 Methodology 5
1.3 Used Technologies 5
2 Algorithms Description 6
2.1 Input Array for Insert Operation 7
2.2 Input Array for Delete Operation 8
2.3 Input Array for Bucket Sort Operation 9
2.4 Input Array for Radix Sort Operation 11
3 User Manual 13
3.1 Single Array Generation 13
3.2 Multiple Arrays Generation 13
3.3 Parameters Definition 13
3.4 Generation Examples 15
Conclusions 16
References 17
License 18

Introduction

There is „Algorithms and Data Structures“ course lectured in the University of Tartu. The course is mandatory for Computer Science curriculum students. One of the things this course teaches is time complexity of an algorithm. Hashtable is widely used as data structure for an algorithm. Insert, delete, bucket sort, radix sort operations are suitable algorithms for hashtable data structure.

During course tests students are given several input arrays of values to apply these algorithms on them using hashtable as data structure. Teacher should generate multiple variants of those input arrays for course test. So each student have their own tasks to solve individually and independently.

Applying algorithms could take significantly different amount of time during course test. It depends directly on number of time consuming operations performed by student solving the task. Input array values should be generated with predefined number of such operations, so everybody are equal in time terms.

Manually generating hundreds of such input arrays is a routine job and should be automated. Especially difficult is to generate completely unassociated ones with same number of time consuming operations. Teacher lacks automation tool for input array generation. Solution was to implement application program for batch generation of those input arrays.

First chapter provides with solution overview: states thesis goal, proposes methodology to be applied and gives insight into technologies used. Second chapter describes classical algorithms and input generation solutions for them using pseudocode from author’s perspective. Third chapter contains user manual for the application with use case examples.

1 Solution Overview

1.1 Thesis Goal

Application user should be able to generate single/multiple input arrays for insert, delete, bucket sort and radix sort operations. Several parameters are available for configuration before result generation. 1-2 parameters directly predefine amount of time consuming operations. Other parameters describe minimum and maximum possible values, size of input array, amount of random arrays to generate. Generated input array/arrays, some extra and debug info can be copied to clipboard.

Main goal was to generate input arrays correctly according to parameters. There is no uniqueness control implemented for batch generated arrays. Probability of similar arrays generation are quite insignificant. Taking that into account neighbour students having similar arrays during test is almost impossible.

1.2 Methodology

First of all classical hashtable algorithms were reviewed for possibility of computer implementation. Picked algorithms were studied to understand what causes difficulties for students performing them. Such time consuming operations were defined for each algorithm.

Parameters directly affecting time consuming operations are called difficulty parameters. Setting these before input generation will establish offered problem difficulty. Other common parameters should not cause significant increase of time needed for student to solve the problem.

Minimum and maximum constraints were set on parameters influencing each other. Afterwards applied input generation solutions were implemented for each described algorithm.

1.3 Used Technologies

Application was written in Java programming language. This is a high-level programming language, which makes developing process easier. “Write once, run anywhere” slogan is utilized resulting in cross-platform software solution. Java is also widely used in the „Algorithms and Data Structures“ course.

Swing framework was used for minimalistic GUI implementation. This is common GUI widget toolkit for Java that does the job. Examples and documentation is available in the internet since Swing is widely used.

Apache Maven was used for build automation and dependency management. This tool is well-known in software development industry and used primarily for Java projects. Running „java -jar .\target\thesis-1.0.0.jar“ after „mvn clean package“ command will launch the application.

2 Algorithms Description

Application generates input arrays for 4 different algorithms in total. Two of them share same generated input array. Delete algorithm uses insert algorithm to define its amount of time consuming operations. Amount of values relocated in hashtable is calculated by comparing result of the first to the second algorithm.

Insert algorithm inserts into empty hashtable generated input array values. Hashtable size should be greater or equal than input array size. Values are inserted in order they are given in input array. Position in hashtable is determined by following formula: $value \bmod hashtableSize$. If position is already occupied by one of the previous values, value is shifted to the right by one position. Shifting is done until value is put into empty position of hashtable. Here shift to the right is time consuming operation of this algorithm. Thus single value can have 0 to $hashtableSize - 1$ position shift. Maximum possible sum of shift of all input array values in hashtable is $count \times (count - 1)/2$.

Delete algorithm inserts into equal-size empty hashtable same input array values without single deleted value. Input array should contain at least one value. Remaining values' positions in hashtable are compared to insert algorithm values' positions. Here value relocation is time consuming operation of this algorithm. Maximum possible amount of relocations is $count - 1$.

Bucket sorting algorithm sorts input values using buckets of hashtable. Hashtable size is chosen equal to input array size. Values are placed into hashtable using hash function $h(k) = m \times (k - a)/(b - a)$. Where k is value, m is array size, a is floor of array minimum value, b is ceiling of array maximum value. Placing evenly distributed input values in buckets should minimize amount of collisions within hashtable buckets. Collision leads to extra sorting operation in the bucket. Each collision means one empty bucket. Here empty bucket is time consuming operation of this algorithm. Maximum possible amount of empty buckets is $count - 2$.

Radix sorting algorithm sorts input values using base of hashtable. Position in hashtable at each step is determined by following formula: $value/base^{step-1} \bmod base$. Then branches from the hastable are concatenated into array. Process repeats with next step until $base^{step-1}$ is greater than array maximum value. Here step is first time consuming operation of this algorithm. Longest branch size is second time consuming operation of this algorithm. There is exactly one branch of that size during all steps. Minimum possible amounts of steps is ceiling of $\log_{base} count$. Maximum possible size of longest branch is $count$.

2.1 Input Array for Insert Operation

Following subchapter describes input generation solution using pseudocode for hashtable insert algorithm.

Parameters:

"*count*" — amount of elements to generate,

"*base*" — hashtable size,

"*start*" — minimum possible element value (inclusive),

"*end*" — maximum possible element value (exclusive),

"*shifts*" — amount of elements shifts to the right during *Insert* operation.

Task: Generate *insertElements* array consisting of *count* elements with values ranging [*start, end*) for *base* cells hashtable with total *shifts* elements' shifts to the right during insert operation.

Solution lies inside "*Insert.solve(parameters)*" method.

1. Define *Element* class as {*value, modulus, position, shift, softDeleted*}.
2. Do following cycle:
    a) Define *usedShifts := 0.*
    b) Define *emptyCells := {0 .. base-1}*.
    c) Define *insertElements* list consisting of *Element* objects:
        i) Generating one by one random distinct values within [*start, end*) range.
        ii) Call *mapToElement(value, base, usedShifts, emptyCells)* method on each *value*.
        iii) Filter each *element* with *notExceedingShifts*(*element, shifts, usedShifts, emptyCells*).
        iv) Stop generating when *count* amount of elements are accepted.
    d) If u*sedShifts == shifts* return *insertElements* list. Generated elements are with sufficient shifts.

notExceedingShifts(*element, shifts, usedShifts, emptyCells*) method:

1. If *usedShifts <= shifts* then return *true*. Element is accepted.
    Else *usedShifts -= element.shift* and add *element.position* to *emptyCells* list. Return *false.*

mapToElement(*value, base, usedShifts, emptyCells*) method:

1. Define *modulus := value % base*.
2. Define *position := modulus*. This is element initial position in hashtable.
3. Define *shift := 0.*
4. Do following cycle:
    a) If *emptyCells* list contains *position*, remove it from the list, increment *usedShifts += shift.*
        Return new *Element(value, modulus, position, shift)* object.
    b) Increment *position++, shift++*. Trying next position, shift to the right is needed.
    c) If *position == base* assign *position := 0*. Position must stay within range *[0, base).*

2.2 Input Array for Delete Operation

Following subchapter describes input generation solution using pseudocode for hashtable delete algorithm.

Parameters:

"*base*" — hashtable size,

"*relocations*" — amount of elements relocated during Delete operation.

Task: Create *deleteElements* array by removing single element from generated *insertElements* array for *base* cells hashtable so *relocations* elements changed their positions during delete operation.

Solution lies inside "*Delete.solve(parameters, insertElements)*" method.

1. Define *shuffledOrder* list as shallow copy of *insertElements* list with shuffled order.
2. For each *randomElement* from *shuffledOrder* list*:*
    a) Mark *randomElement.softDeleted := true.*
    b) Define *deletedElements* list as call to
        convertFromActiveElementValues(*insertElements, base*) method.
    c) Define *relocatedValues* list as call to
        findRelocatedValues (*deleteElements, insertElements*) method.
    d) If size of *relocatedValues* list equals to *relocations* parameter return *deletedElements* list.
    e) Mark *randomElement.softDeleted := false.*
3. Return empty list and start over with *Insert* algorithm. Impossible to delete element from insertElements list achieving desired amount of elements relocated.

convertFromActiveElementValues(*insertElements, base*) method:

1. Define *usedShifts := 0.*
2. Define *emptyCells := {0 .. base-1}*.
3. Remove from *insertElements* list single element marked *softDeleted.*
4. Call *mapToElement(value, base, usedShifts, emptyCells)* method on each *value*.
5. Return *insertElements* list.

findRelocatedValues(*deleteElements, insertElements*) method:

1. Define *relocatedValues* list.
2. For each *delElem* from *deleteElements* list:
    a) For each *insElem* from *insertElements list:*
        i) If *delElem.value == insElem.value* and *delElem.position != insElem.position*:
            Add *delElem.value* to *relocatedValues* list.
3. Return *relocatedValues* list.

2.3 Input Array for Bucket Sort Operation

Following subchapter describes input generation using pseudocode for hashtable bucket sort algorithm.

Parameters:

"*count*" — amount of elements to generate (also total amount of buckets),

"*minIncl*" — minimum possible element value (inclusive),

"*maxExcl*" — maximum possible element value (exclusive),

"*complexity*" — amount of empty buckets.

Task: Generate array consisting of *count* elements with decimal values ranging [*minIncl, maxExcl*) for *count* cells hashtable with *complexity* empty buckets during bucket sort operation.

Note: $(b-a)/m$ must be within {1, 2, 5, 10, 20} since $h(k) = m \times (k-a)/(b-a)$

Solution lies inside "*BucketSort.solve(parameters)*" method.

1. Define *rangeLimit := maxExcl - minIncl*. This is maximum possible range.
2. Define possible *dividers := {1, 2, 5, 10, 20}*. Those are suitable for quick calculation by students.
3. Remove *div* from *dividers* such as *div * count > rangeLimit*. Dividers are sizes of the buckets.
4. Pick random *divider* from the remaining *dividers*.

5. Define *range := count * divider*. (This is also called *b-a*)
6. Pick random *start* within *[minIncl, maxExcl - range]*. (This is also called *a*)
7. Define *end := start + range*. (This is also called *b*)

Note: *h(k) = m * (k - a) / (b – a).*

8. Generate *emptyBuckets* list of distinct values within *[1, count - 1).*
   List size is equal to *complexity* parameter.
9. Define *requiredBuckets* list of all values within *[0, count)* excluding *emptyBuckets* values.
10. Define *bucketToElementsAmount* map for holding *bucket->elements-amount* as key->value pairs.

Maximum 10 elements can be in single bucket since 10^-1 precision is used.

11. Define *buckets* non-distinct list. Add *requiredBuckets* values to it immediately.
12. Do following cycle:
    a) Generate *randomBucket* within *[0, count).*
    b) Proceed to a) if *emptyBuckets* contains this *randomBucket*.
    c) Increment value++ in *bucketToElementsAmount* map for *randomBucket* key.
    d) Proceed to a) if value in *bucketToElementsAmount* map for *randomBucket* key is greater than *9*.
    e) Add this *randomBucket* to *buckets* list.
    f) Break cycle when *buckets* size reaches *count* parameter.

13. Define *elements* list.
14. Define *headNeeded := true, tailNeeded := true*.

    They are used to guarantee head and tail elements presence.

    Those element values are needed so students can calculate properly *[a, b)* range.
15. For each *bucket* from *buckets* list:

    a) Define *element* variable.

    b) Do following cycle:

        i) Generate *randomFraction* within *{0.0, 0.1, 0.2, 0.3, 0.4, 0.5, 0.6, 0.7, 0.8, 0.9}*.

        ii) If *headNeeded* is *true* and bucket is *0*:

            Mark *headNeeded* := *false*, assign *element := start + randomFraction*.

            So there will be element within *[start, start+1)*.

        iii) Else if *tailNeeded* is *true* and *bucket* is *count - 1*:

            Mark *tailNeeded := false, element := end - 1 + randomFraction*.

            So there will be element within *[end-1, end)*.

        iv) Else pick random *randomWhole* within *[0, divider)*.

            *element := start + bucket * divider + randomWhole + randomFraction*.

        v) Break cycle if *elements* does not contain this *element*. Duplicates are not allowed.

    c) Add *element* to *elements* list.

16. Shuffle *elements* list.
17. Define *generatedArray* string as concatenating *elements* list using ' ' as delimeter.
18. Define *firstRow* string as concatenating sorted *emptyBuckets* list using ' ' as delimeter.
19. Define *secondRow* string providing extra info "[*start, end*) => divider := (b-a)/m = (*end-start*)/*count"*.
20. Return "*Result(generatedArray, firstRow, secondRow)"*.

2.4 Input Array for Radix Sort Operation

Following subchapter describes input generation using pseudocode for hashtable radix sort algorithm.

Parameters:

"*count*" — amount of elements to generate,

"*base*" — or the radix (also hashtable size),

"*steps*" — amount of sorting steps taken,

"*longestBranchSize*" — amount of elements in longest branch.

Task: Generate array consisting of *count* elements for *base* cells hashtable
with *longestBranchSize* elements in longest branch during *steps* step radix sort operation.
Array values must be within [0, *base^steps*). Should be value within [*base^(steps-1)*, *base^steps*).

Solution lies inside "*RadixSort.solve(parameters)*" method.

1. Define *start := base^*(*steps* - 1) and *end := base^steps*.
2. Pick random *randomStep* within [0, *steps*). This determines longest branch step.
3. Define *divider := base^randomStep.* This determines longest branch divider.
4. Pick random *randomModulus* within [0, *base*). This determines longest branch modulus.

5. Define *branchSizes* map for holding *step+"_"+ modulus->branchSize* as key->value pairs.
   It will hold branch sizes at each step for every modulus. There will be exactly one value equal to *longestBranchSize* in this map at *randomStep+" "+randomModulus* key.
   All the rest values must be less than *longestBranchSize.*

getKeys(*base, steps, longestBranchSize, element, branchSizes, longestStep*) method:

1. Define *keys* list.
2. For each *step* within {0 .. *steps* - 1}:
   a) Define *divider := base^step.*
   b) Define *modulus := element / divider % base*.
   c) Define *key := step+"_"+modulus*.
   d) Define *branchSize* as value from *branchSizes* mapped by *key.*
   e) If *step == longestStep* define *branchSizeLimit := longestBranchSize.*
      Else define *branchSizeLimit := longestBranchSize – 1*.
      Always *false* if *longestStep* was assigned *steps* value.
      Only once *true* if *longestStep* was assigned *randomStep* value.
   f) If *branchSize* < *branchSizeLimit* add *key* to *keys* list. Element will be accepted.
      Else method returns empty list. Element will not be accepted.
3. Method returns *keys* list for *branchSizes* map.

6. Define *elements* list.

7. Define *cycleRuns :=* 0.

8. Do following cycle:

    a) If *cycleRuns == longestBranchSize* define *element* within [*start, end*).

        Else define *element* within [0*, end*). This secures that *elements* list has big element

        so proper amount of sorting steps should be taken.

    b) Define *isGeneratingLongest := cycleRuns < longestBranchSize.*

        This means generation of longest branch elements first.

    c) Proceed to a) if *isGeneratingLongest* is *true* and *element / divider % base != randomModulus.*

        Generated longest branch *element* does not suit predefined longest branch modulus.

    d) Proceed to a) if *elements* list contains *element*. Duplicates are not allowed.

    e) Define *longestStep := isGeneratingLongest ? randomStep : steps.*

    f) Define *keys* list as call to *getKeys(base, steps, longestBranchSize, element, branchSizes, longestStep).*

        It returns keys for *branchSizes* map values to increment.

    g) Proceed to a) if *keys* list is empty. Branch size limit is exceeded, element is not accepted.

    h) Increment value++ in *branchSizes* map for each *key* in *keys* list. Element is accepted.

    i) Add *element* into *elements* list.

    j) Increment *cycleRuns*++.

    k) Break cycle when *elements* size reaches *count* parameter.

9. Define *longestBranch* string as concatenating first *longestBranchSize* values

    from *elements* list using ' ' as delimeter.

10. Shuffle *elements* list.

11. Define *generatedArray* string as concatenating *elements* list using ' ' as delimeter.

12. Define *secondRow* string providing extra info “step: *step* + 1+. => divider: *divider* => modulus: *modulus* => mask: *step*+_+*modulus”*.

13. Return “*Result(generatedArray*, *longestBranch, secondRow)”*.

3 User Manual

Minimalistic GUI consists of 2 tabs „Single“ and „Batch“. They are used for single and multiple array generation respectively.

3.1 Single Array Generation

Select control at „Single“ tab allows switching between 3 generation tasks: „InsertDelete“, „BucketSort“, „RadixSort“. Algorithm related parameter spinners are shown once generation task is selected. Parameter spinners are limited by minimum and maximum bounds.

Generation of input array is started by clicking “Generate array” button. Several restrictions are applied on parameters’ values combination so that generation could be possible. Generation does not start and error text is shown to the user if one of the listed conditions occurs.

Generated result consists of input array values, extra and debug info. It is shown in separate text fields. Each text field value can be copied to clipboard by “Copy to clipboard” button.

3.2 Multiple Arrays Generation

Each selected generation task with defined parameters can be expanded at “Batch” tab. Single parameter spinner allows defining amount of random arrays to generate.

Generation of input arrays is started by clicking “Generate” button. Same parameter restrictions are applied and error text is shown as in single array generation.

Generated result consists of input arrays’ values. It is shown in separate text area. All arrays can be copied to clipboard by “Copy all” button.

3.3 Parameters Definition

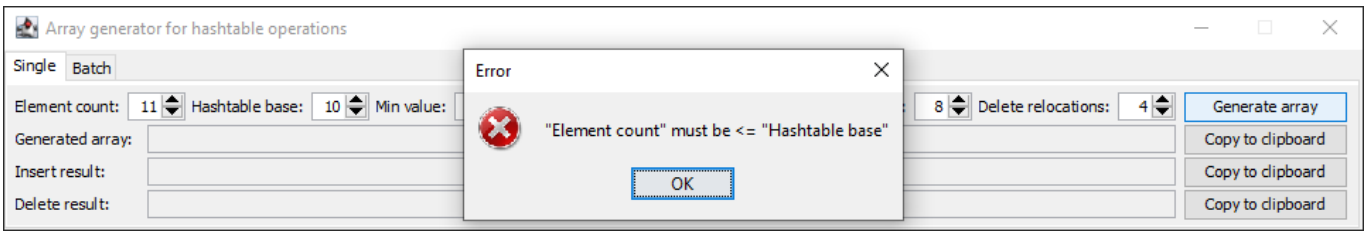
Parameter spinners:

Label Text	**Name**	**Default**	**Range**	**Algorithm**
„Element count“	*count*	10	[1, 30]	InsertDelete BucketSort RadixSort
„Hashtable base“	*base*	10	[2, 20]	InsertDelete RadixSort
„Min value“	*minValue*	0	[-99, 99]	InsertDelete BucketSort
„Max value“	*maxValue*	20	[-99, 99]	InsertDelete BucketSort
„Insert shifts“	*shifts*	8	[0, 25]	InsertDelete
„Delete relocations“	*relocations*	4	[0, 15]	InsertDelete
„Empty buckets“	*emptyBuckets*	5	[0, 18]	InsertDelete BucketSort
„Steps“	*steps*	2	[1, 10]	InsertDelete RadixSort
„Longest branch“	*longestBranch*	3	[1, 30]	InsertDelete RadixSort
„ random arrays“	*quantity*	10	[2, 50]	InsertDelete BucketSort RadixSort

Parameter restrictions:

Condition	**Error Text**	**Algorithm**
count > base	„Element count“ must be <= „Hashtable base“	InsertDelete
maxValue - minValue < count	„Max value“ - „Min value“ must be >= „Element count“	InsertDelete
*shifts > (count - 1) * count / 2*	„Insert shifts“ must be <= („Element count“ - 1) * „Element count“ / 2	InsertDelete
relocations > count - 1	„Delete relocations“ must be <= „Element count“ - 1	InsertDelete
maxValue – minValue < count	„Max value“ – „Min value“ must be >= „Element count“	BucketSort
emptyBuckets > count - 2	„Empty buckets“ must be <= „Element count“ - 2	BucketSort
*count > (count - emptyBuckets) * 10*	„Element count“ must be <= („Element count“ - „Empty buckets“) * 10	BucketSort
count > base^steps	„Element count“ must be <= „Hastable base“ in power of „Steps“	RadixSort
longestBranch > count	„Longest branch“ must be <= „Element count“	RadixSort
longestBranch > base^(steps-1)	„Longest branch“ must be <= „Hashtable base“ in power of („Steps“ - 1)	RadixSort

Error example:



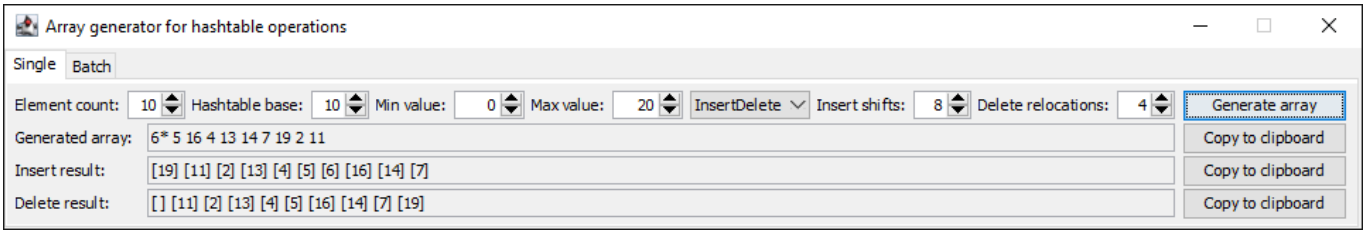
3.4 Generation Examples

“InsertDelete” task with default parameter values produces the following results:

“Generated array:” 6 5 16 4 13 14 7 19 2 11*

“Insert result:” [19] [11] [2] [13] [4] [5] [6] [16] [14] [7]

“Delete result:” [ ] [11] [2] [13] [4] [5] [16] [14] [7] [19]

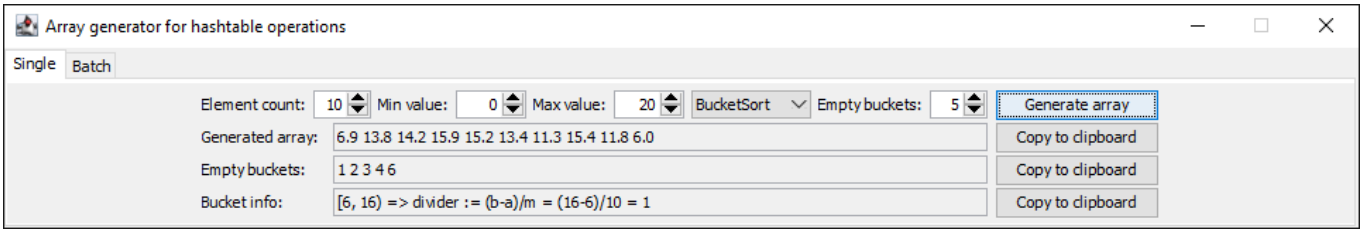


“BucketSort” task with default parameter values produces the following results:

“Generated array:” 6.9 13.8 14.2 15.9 15.2 13.4 11.3 15.4 11.8 6.0

“Empty buckets:” 1 2 3 4 6

“Bucket info:” [6, 16) => divider := (b-a)/m = (16-6)/10 = 1

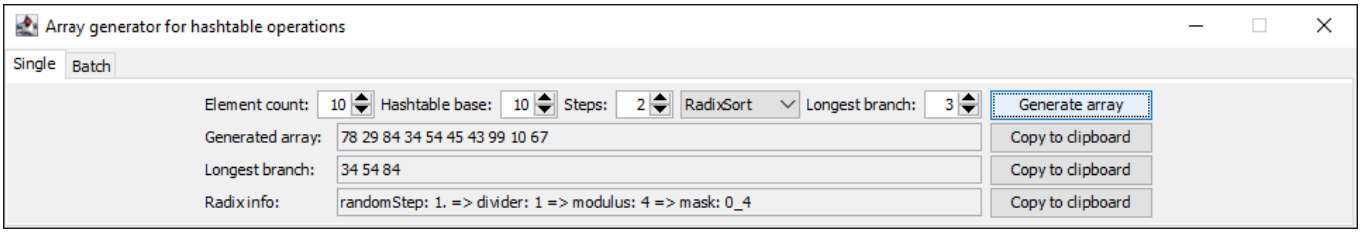


“RadixSort” task with default parameter values produces the following results:

“Generated array:” 78 29 84 34 54 45 43 99 10 67

“Longest branch:” 34 54 84

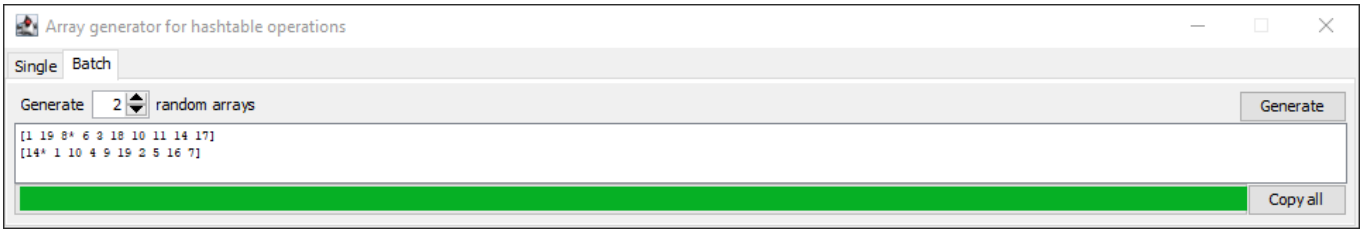
“Radix info:” randomStep: 1. => divider: 1 => modulus: 4 => mask: 0_4



“InsertDelete” batch task with default parameter values produces following results:

[1 19 8 6 3 18 10 11 14 17]*

[14 1 10 4 9 19 2 5 16 7]*



Conclusions

Thesis goal was achieved since input generation solutions were composed and implemented. Time consuming operations performed by students for insert, delete, bucket sort and radix sort algorithms were studied and pointed out. Application correctly generates results in reasonable amount of time for end user. Simple graphical user interface was created to make program clear and demonstrative.

Application can be used for educational purposes during „Algorithms and Data Structures“ course. It offers interactivity for students performing these classical algorithms. Proposed applied solution is cross-platform thus allows other interested parties to use it in their projects.

Classical hashtable algorithms were covered meaning current thesis topic is complete. Input generation for classical tree and graphs algorithms are other possibilities for thesis topic. Application is planned to be integrated into prospective DeepMOOC learning platform. Laboratory for Software Science develops DeepMOOC within Institute of Computer Science at University of Tartu.

References

[1] J. Kiho. Algoritmid ja andmestruktuurid. 2003. https://moodle.ut.ee/pluginfile.php/76788/mod_resource/content/3/LOENG2011/ads2003.pdf (09.05.2022)

[2] Algoritmid ja andmestruktuurid (LTAT.03.005). Läbimänguslaidid. 2022. https://moodle.ut.ee/mod/book/view.php?id=754639&chapterid=47479 (09.05.2022)

[3] Samuel Johannes Pitko. Input Generation for Array Algorithms. 2021. https://comserv.cs.ut.ee/ati_thesis/datasheet.php?id=72137 (09.05.2022)

[4] Apache Maven Project. https://maven.apache.org/ (09.05.2022)

License

Non-exclusive licence to reproduce the thesis and make the thesis public

I, **Nikolai Voitsehhovski**,

1. grant the University of Tartu a free permit (non-exclusive licence) to reproduce, for the purpose of preservation, including for adding to the DSpace digital archives until the expiry of the term of copyright, my thesis

Input Generation for Hashtable Algorithms,

supervised by **Ahti Põder, Ph.D**.

2. I grant the University of Tartu a permit to make the thesis specified in point 1 available to the public via the web environment of the University of Tartu, including via the DSpace digital archives, under the Creative Commons licence CC BY NC ND 4.0, which allows, by giving appropriate credit to the author, to reproduce, distribute the work and communicate it to the public, and prohibits the creation of derivative works and any commercial use of the work until the expiry of the term of copyright.

3. I am aware of the fact that the author retains the rights specified in points 1 and 2.

4. I confirm that granting the non-exclusive licence does not infringe other persons' intellectual property rights or rights arising from the personal data protection legislation.

Nikolai Voitsehhovski

12/05/2022